PERQ
Systems
Corporation

PERQ FILE SYSTEM

March 1984

This manual is for use with POS Release G.5 and subsequent releases until further notice.

PERQ Systems Corporation
2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

PREFACE

This document describes the file system for the PERQ Operating System.

The POS file system provides a directory oriented structure that allows you to store and manipulate files on hard disks and floppy disks. The system supports both hard disks (Shugart 12- and 24-k byte, Micropolis 35-k byte, and 5.25") and floppy disks (single or double sided).

The hardware addressing of devices occurs through microcode, using a distinct address. Software addresses the devices through the microcode using a standard address. Chapter 1 describes the distinct (physical) and standard (logical) addresses. The Chapter also describes the logical structures that provide the file system interface between physical and logical structures.

Chapter 2 details the data structures of the logical structures. This Chapter describes the significance of each word in the file system logical structures.

Chapter 3 defines the format of two important types of files: segment (.Seg) files produced by a compiler; and directory (.DR) files that form the keystone of the file system. This Chapter also describes the method in which files are entered or listed in the directories. The Chapter does not attempt to define the format of executable (.Run) files nor system bootstrap (.Boot) files. (The module Code.Pas defines .Run file format and the MakeBoot utility writes .Boot files.)

Chapter 4 provides a step-wise refinement through the fundamental file system operations that allow you to create, read, write, delete, and close files. Other functions require one or more of these fundamental operations.

Chapter 5 describes the file system utilities Partition, Scavenger, MakeBoot, and Fixpart. The Partition utility initializes a disk for use with the file system; it creates the logical structures. The Scavenger utility provides disk maintenance. The MakeBoot utility permits you to create new, bootable software systems. Finally, the Fixpart utility provides last resort maintenance of disks; its use is seldom required and, unless you have a firm grasp on the file system concepts, always discouraged.

Note that on a PERQ system, the low-order word precedes the high-order word in memory. All values represented in this manual are in memory order. Bits are numbered as follows:

```
     High-byte          |     Low-byte
15 14 13 12 11 10 9 8   | 7 6 5 4 3 2 1 0
```

The disk addresses represented in this manual are octal values.

CHAPTER 1

INTRODUCTION

The syntax of POS file names reflects the hierarchical organization of the POS file system as follows:

```
device:partition>directory>filename
```

The mass storage devices (hard or floppy disk) form the base of the hierarchy. The file system divides each device into a number of sections, known as partitions. Each partition can contain any number of directories. Directories are special format files that can contain names and addresses of files as well as other directories; you can nest directories and thus form a multi-level directory structure. In the POS file system, all files can be noncontiguous. However, files, including directories, cannot cross partition boundaries; portions of files can only be scattered throughout the partition in which they reside.

The sections that follow discuss the format and composition of the POS file system structure.

1.1 PHYSICAL FORMAT OF DEVICES

There are two types of mass storage devices: hard disks and floppy disks. Each device consists of discrete positions known as cylinders. The device's read/write heads divide the cylinders into tracks. Each track is divided into sectors. Table 1-1 lists the number of tracks per cylinder and the number of sectors per track for Micropolis and Shugart hard disks and floppy disks.

Throughout this chapter statistics are not given for 5.25" disks, as the numbers vary depending on the manufacturer. If you purchased the disk from PERQ Systems, the file Disk.Params on your machine will contain information about the disk.

Table 1-1
Cylinders, Tracks, and Sectors for Volumes

Drive/ capacity	Cylinders	Tracks/ cylinder	Sectors/ track
Micropolis35 35-MB	580 (0 - 579)	5 (0 - 4)	24 (0 - 23)
Shugart 4000 12-MB	202 (0 - 201)	4 (0 - 3)	30 (0 - 29)
Shugart 4002 24-MB	202 (0 - 201)	8 (0 - 7)	30 (0 - 29)
Single-sided floppy	77 (0 - 76)	1	26 (1 - 26)
Double-sided floppy	77 (0 - 153)	2	26 (1 - 26)

The Shugart disks consist of 202 cylinders numbered from 0 through 201. A 12-MB Shugart disk uses four read/write heads and thus contains four tracks per cylinder, numbered 0 through 3. A 24-MB Shugart disk simply doubles the number of read/write heads and thus contains eight tracks per cylinder (tracks 0 through 7). The tracks on a Shugart disk are divided into 30 sectors, numbered 0 through 29.

Figure 1-1 illustrates the cylinder, track, and sector divisions of a 12-MB Shugart disk.

The Micropolis disk consists of 580 cylinders numbered from 0 through 579 and uses five read/write heads. Thus, a Micropolis disk contains five tracks per cylinder (tracks 0 through four). The tracks on a Micropolis disk are divided into 24 sectors, numbered 0 through 23.

Figure 1-2 illustrates the cylinder, track, and sector divisions of a 35-MB Micropolis disk.

The floppy disk consists of 77 cylinders numbered from 0 through 76. A single-sided floppy disk uses one read/write head and thus contains one track per cylinder. A double-sided floppy disk uses two read/write heads (one on each side) and thus contains two tracks per cylinder (tracks 0 and 1). The tracks on a floppy disk are divided into 26 sectors, numbered 1 through 26.

Figure 1-3 illustrates the cylinder, track, and sector divisions of

a double-sided floppy disk.

Figure 1-1
12-MB Shugart Disk Organization

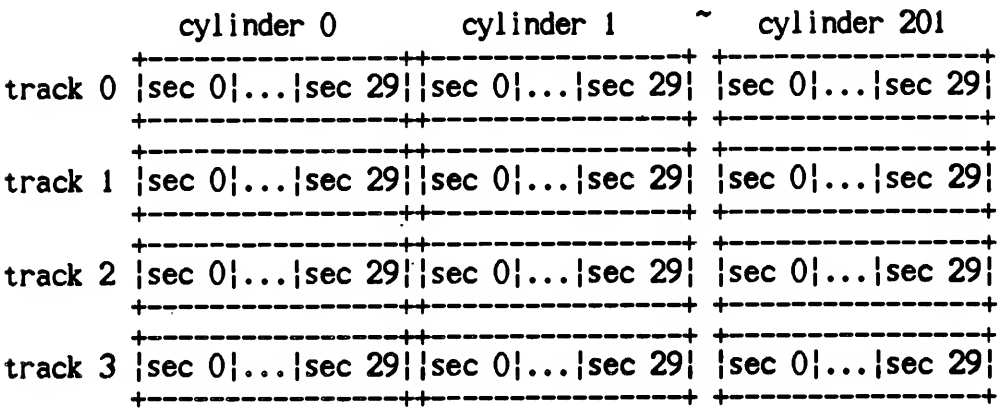


Figure 1-2
35-MB Micropolis Disk Organization

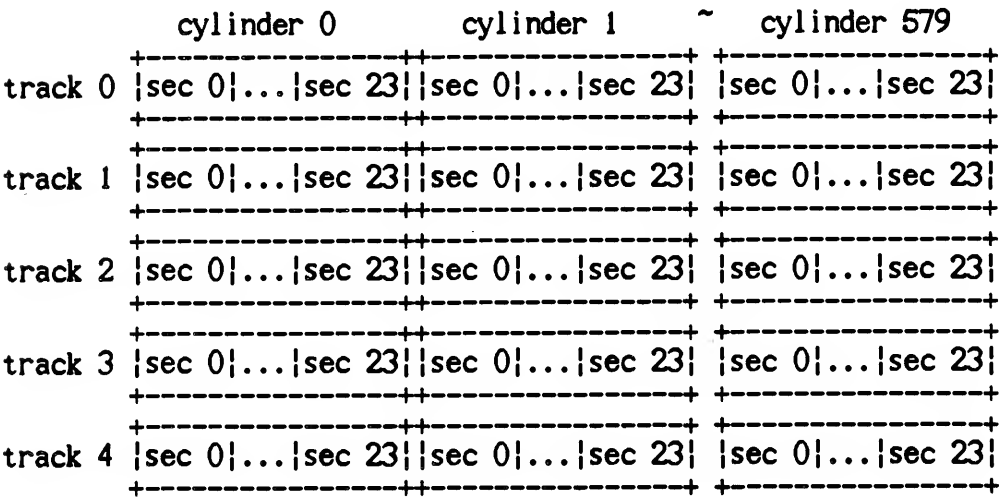
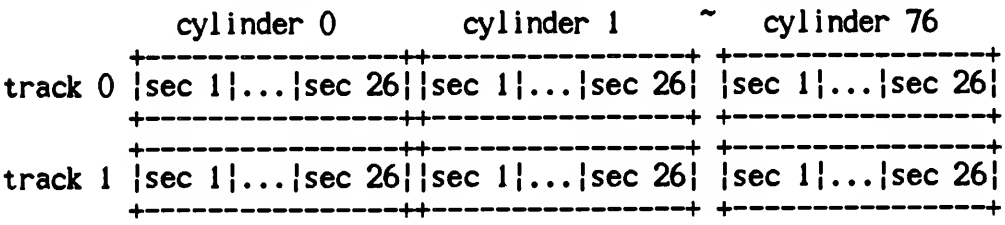


Figure 1-3
Double-sided Floppy Disk Organization



Each sector on a device minimally contains a data block and a three-word (6-byte) physical header.

As its name implies, a data block is the data area of a sector. On a hard disk, the data block is an array of 256 16-bit words that contain data. On a floppy disk, the data block is an array of 64 16-bit words that contain data.

The physical header permits the disk controller to verify head positioning and uniquely identifies each sector on the device; it contains the cylinder number, the track number within the cylinder, and the sector number within the track.

Physical Disk Addresses (PDAs) specify the exact physical location of a sector; the PDA defines sector location by cylinder number, track number within that cylinder, and sector number within that track. To communicate with a device, microcode passes the PDA to the specific hard disk or the floppy disk controller.

A PDA is a 32-bit (2-word) value. The Shugart disk controllers expect the cylinder, track, and sector specification in a single word while the Micropolis, 5.25" disks, and floppy disk controllers expect the specification in two words.

In a Shugart disk PDA, the low order word specifies the cylinder, track, and sector number as an octal value. The high byte (bits 8 - 15) of the low order word contains the cylinder number. Bits 5 through 7 of this word contain the track number. Bits 0 through 4 contain the sector number. Thus, the first Physical Disk Address (expressed in octal) of a Shugart disk is:

```
  Low   High
000000 000000 (specifies cylinder 0, track 0, sector 0)
```

The highest Physical Disk Address for a 12-MB Shugart disk is:

```
  Low   High
144575 000000 (specifies cylinder 201, track 3, sector 29)
```

The highest Physical Disk Address for a 24-MB Shugart disk is:

```
  Low   High
144775 000000 (specifies cylinder 201, track 7, sector 29)
```

The high order word of a Shugart disk PDA is not significant, but must be all zeroes.

In a Micropolis disk PDA, both the low and high order words are significant. The low byte (bits 0 - 7) of the low order word specifies the sector and the high byte (bits 8 - 15) of the low

order word specifies the track. The high order word specifies the cylinder number. Thus, the first Physical Disk Address (expressed in octal) of a Micropolis disk is:

```
  Low   High
000000 000000 (specifies cylinder 0, track 0, sector 0)
```

The highest Physical Disk Address for a 35-MB Micropolis disk is:

```
  Low   High
001103 002427 (specifies cylinder 579, track 5, sector 23)
```

In a floppy disk PDA, the low-order word specifies the sector and the high-order word specifies the cylinder. Note that the floppy disk controller can infer, directly, the track number from the cylinder number and thus does not require a track number specification. The first Physical Disk Address (expressed in octal) of a floppy disk is:

```
  Low   High
000001 000000 (specifies cylinder 0, track 0, sector 1)
```

The highest Physical Disk Address for a single-sided floppy disk is:

```
  Low   High
000032 000114 (specifies cylinder 76, track 0, sector 26)
```

The highest Physical Disk Address for a double-sided floppy disk is:

```
  Low   High
000032 000231 (specifies cylinder 76, track 1, sector 26)
```

On a hard disk, each sector also contains a separate 8-word data area, known as a logical header. This area is used by the POS file system to verify to which file the data belongs and to permit a recovery operation in the event of a failure. The file system maps the logical header structure onto a floppy disk. Section 1.2 describes the logical header.

1.2 LOGICAL FORMAT OF DEVICES

The microcode views a disk as a cluster of uniquely addressable sectors. Since the disks are different, the microcode provides different cylinder, track, and sector abstractions of the devices. Conversely, the file system views a disk simply as an array of data area pairs addressed by sequential integers in the range 0 through n. (Note that this view of a disk is implicit and does not directly correspond to any module or procedure in the file system.)

The view of a disk as a sequential array of data area pairs permits the file system to treat hard and floppy disks uniformly.

The file system refers to the data area pairs as a logical block. A logical block consists of an eight-word logical header and a 256-word data block.

The hard disk hardware supports the data area pair concept directly. Thus, implementation of logical blocks (logical header/data block) is straightforward on hard disks; each physical sector on a hard disk that is accessible to the file system can be viewed as a logical block.

The implementation of the data area pair concept on a floppy disk is more difficult; floppy disk hardware only supports a 64-word data block. To map the logical block concept onto a floppy disk, the file system uses the first sector of each track as the logical header portion of the data area pair and combines four 64-word floppy sectors to form the 256-word data area of the pair. Thus, each track on a floppy disk contains six logical blocks. The first sector of each track contains the logical headers for the track's six logical blocks. Note that the logical blocks begin with the third sector since the file system never uses the second sector on a floppy disk.

The file system assigns a sequential number to each logical block on a disk, known as the Logical Block Number or LBN. The logical blocks on a disk are numbered consecutively from 0 to n-1, where the disk contains n logical blocks. A Logical Block Number identifies each data area pair on a disk that is accessible to the file system.

A file system volume contains a microprogram that runs diagnostics and reads the .Boot and .MBoot files (the .Boot file contains the operating system and the .MBoot file contains the QCode interpreter and IO microcode). This microprogram resides on disk in physical space reserved for it, known as the boot area. The boot microcode in ROM accesses the reserved area to initiate the boot sequence. Once the system boots, only the MakeBoot utility (Section 5.3 describes the MakeBoot utility), which writes the microprogram, should access this area. The code is maintained outside the logical disk structures and is not accessible to the file system.

On a Shugart disk, the first track (cylinder 0, track 0, sectors 0 through 29) is reserved for the bootstrap code. Therefore, the first logical block on a Shugart disk, LBN 0, is the first physical sector following the boot code (cylinder 0, track 1, sector 0). Thus, Shugart disk LBNs range from 0 through n-1 where n is the maximum number of sectors on the disk minus 30.

On a Micropolis disk, the first two tracks (cylinder 0, tracks 0 and 1) are reserved for the bootstrap code. Therefore, the first

logical block on a Micropolis disk, LBN 0, is at physical location cylinder 0, track 2, sector 0. Thus, Micropolis disk LBNs range from 0 through n-1 where n is the file system's maximum number of sectors on a disk minus 48.

On a floppy disk, five cylinders (cylinders 0 through 4) are reserved for the bootstrap code. Therefore, the first logical block on a floppy disk, LBN 0, starts at cylinder 5, track 0, sector 3. The LBNs range from 0 through 432 on a single-sided floppy and from 0 through 888 on a double-sided floppy.

The file system supports a single address space for the logical blocks on both hard and floppy disks, known as a Logical Disk Address or LDA.

Like a Physical Disk Address, Logical Disk Addresses are 32-bit (2-word) values, but the values are distinctly different. A PDA uniquely identifies a sector on a given device by cylinder number, track number within the cylinder, and sector number within the track. An LDA encodes a disk specifier (hard or floppy) and an positive number representing the disk specific Logical Block Number.

Both words of an LDA are significant for both hard and floppy disks. The following describes the significance of each bit that forms an LDA.

Low word:

Bits 0 through 7 are reserved for future use. The present design reserves the bits to specify the desired word within the logical block's data area. Future PERQ operating systems will use the bits, but not for word references. Under the current system, these bits must be zero so that they can be made non-zero for future use. Since they are reserved for future use, programs reading these bits should not assume that the bits are in fact zero.

Bits 8 through 15 form the low eight bits of the LBN.

High word:

Bits 0 through 10 form the high eleven bits of the LBN.

Bits 11 through 13 specify the volume (000 indicates hard, 100 indicates floppy).

Bits 14 and 15 indicate whether the address is on disk, and therefore in permanent storage, or in virtual memory. Under the current system, these bits are always set to indicate permanent storage.

The module DiskUtility.Pas contains functions which convert an LDA to a PDA or an LBN, a PDA to an LDA, and an LBN to an LDA (perform conversions through DiskIO.pas, which calls DiskUtility.Pas). The IO system provides the means for microcode and Pascal to communicate at the cylinder, track, sector abstraction level.

Most I/O operations must know the contents of the block's logical header. The logical header contains the following information:

File serial number (2 words)
Logical block number (1 word)
Filler word (1 word)
Next Logical Disk Address (2 words)
Previous Logical Disk Address (2 words)

The serial number is the Logical Disk Address of the File Information Block (FIB). Section 2.4 describes the FIB in detail. Note that the serial number is written as a Physical Disk Address and converted to an LDA when the header is read.

The Logical block number in the logical header represents the block number within a series of blocks, not the sequential LBN of the block. This manual refers to the logical block number of a block within a series of blocks as the relative logical block number.

The filler word is used by the file system in its free block allocation scheme. When a block is part of the free list, the filler word in its logical header contains the LBN of the first block of the list. When the block is allocated, the filler word is unchanged.

The Next Logical Disk Address and the Previous Logical Disk Address form doubly linked lists of blocks. The values are the LDAs of the next block and the previous block of a series of blocks. For both pointers, a zero value terminates the link. The values for these pointers are actually written as Physical Disk Addresses and converted to LDAs when the header is read for processing purposes.

To access a logical block, the block's LBN is first extracted from the LDA and then converted into a cylinder, track, sector specification. For example, assume the file system must access a logical block on a 24-MB Shugart disk at the following Logical Disk Address:

151000 140000

First, it extracts the Logical Block Number. The high byte of the low order word forms the low eight bits of the LBN.

Low-order word
151000
high-byte|low-byte

11010010¦00000000

Bits 0 through 10 of the high order word form the high eleven bits of the LBN.

High-order word
140000
high-byte¦low-byte
11000000¦00000000

Thus, the binary representation of the LBN is:

00000000011010010

which converts to 322(8) or LBN 210. Since the numbering scheme for logical blocks on a Shugart disk omits the first 30 sectors, Shugart disk LBN 210 is actually physical sector 240. The cylinder, track, sector specification for physical sector number 240 is cylinder 1, track 0, sector 0 (derived from 30 sectors per track and 8 tracks per cylinder).

1.3 DEVICE COMPOSITION

The device that carries the file system structure is referred to as a volume. A volume is simply an ordered set of logical blocks.

A Device Information Block (DIB) identifies a volume as a POS file structured volume. The DIB is the first logical block and has a defined physical location on the volume. It contains all the fixed information about the volume; the DIB describes the number of logical blocks on the volume, specifies the physical addresses for the Qcode and microcode boots, and identifies the volume with a label (an ASCII string of one to eight characters). Thus, the DIB serves as the foundation of the volume structure. Section 2.2 describes the DIB in detail.

The file system divides the set of logical blocks on a volume into physically contiguous sections known as partitions. You create and modify partitions on a volume using the Partition program. (The Partition program accepts input for partition name, size, and permits you to specify the number of partitions. Section 5.1 provides complete details on the Partition program.) The DIB contains pointers to the first block of each physically contiguous disk area.

The first block of each partition is the Partition Information Block (PIB), which details partition specific information. Section 2.3 describes the PIB in detail.

Any data of interest in a partition (that is all blocks not available for allocation) are contained in segments. A segment is

a cluster of logical blocks linked together by pointers in each block's logical header. Each segment in a partition has a block that describes the segment, known as the information block. Section 2.4 describes the information block in detail. This block specifies the type of segment and, most importantly, contains pointers to the blocks that form the segment. All blocks that form the segment contain the Logical Disk Address of the information block in the serial number of their logical header. Thus, the LDA of the information block identifies all blocks that belong to a given segment. The file system refers to the LDA of a segment's information block as the SegID. The SegID is used in all references to a segment.

Named segments form files. A file name consists of four portions and takes the form:

device:partition>directory1>...directory9>filename

A complete file specification is referred to as a full filename; it specifies the complete path for a file. Since the file system applies defaults for device, partition, and directories, users need not specify a full filename. A file specification that omits the device and partition portions (includes only directories and filename) is referred to as a partial file specification. A file specification that omits all portions except the file's name is referred to as a simple file specification.

Each file in a partition has an entry in a directory and the information block from the segment. The directory entry contains a pointer to the information block of the file. Section 3.5 describes directory entries in detail. The file's information block contains the same type and pointer data as the information block from a segment with additional file system specific information. Thus, all files are segments, but since a file has a directory entry and additional information in its information block, not all segments are files.

The information block of a segment and the information block of a file share the same data structure, known as the File Information Block or FIB. Section 2.4 describes the FIB in detail.

Since a segment is the origin of a file, the serial number in the logical header of all blocks that form the file contains the LDA of the FIB (the SegID). However, the file system refers to files by FileID.

For hard disks, a FileID is the low-order 16 bits of the LBN of the FIB (the LBN extracted from the SegID). For floppy disks, a FileID is the LBN of the FIB plus 160000(8) (the LBN extracted from the SegID logically anded with 160000(8)). Thus, FileIDs are unsigned 16-bit integers (cardinal numbers) ranging from 0 through 64K. (The cardinal numbers from 0 through 56K represent hard disk LBNs,

from 56K+1 through 64K-1 represent floppy disk LBNs.)

The POS file system provides directories to allow the organization of files in a meaningful and recognizable way. Each partition on the volume can contain any number of directories. While the address of the File Information Block is sufficient to locate a file uniquely in a partition, the address is hardly mnemonic. A directory is a file that contains the names and pointers to the FIBs of each file within the directory; directories associate symbolic names with the address of the FIB.

Since directories are simply files that contain pointers to other files, the files contained in a directory can be other directories. Thus, directories can be nested to form a hierarchical, multi-directory structure. You can construct directory hierarchies of arbitrary depth and complexity to structure files in whatever manner is convenient.

The main directory in a partition (the base of the multi-directory structure) is the root directory for that partition. Therefore, each partition on the volume contains at least one directory, Root.DR. Subsequent levels are referred to as subdirectories. Note that since a directory is a file, an FIB exists for each directory.

CHAPTER 2

FILE SYSTEM DATA STRUCTURES

This chapter describes the format of the logical structures on a file system volume.

2.1 OVERVIEW

The Device Information Block (DIB) identifies a file structured volume and contains pointers to Partition Information Blocks (PIBs).

Each PIB defines the limits of physically contiguous areas on the volume and contains a pointer to the File Information Block (FIB) of each partition's root directory.

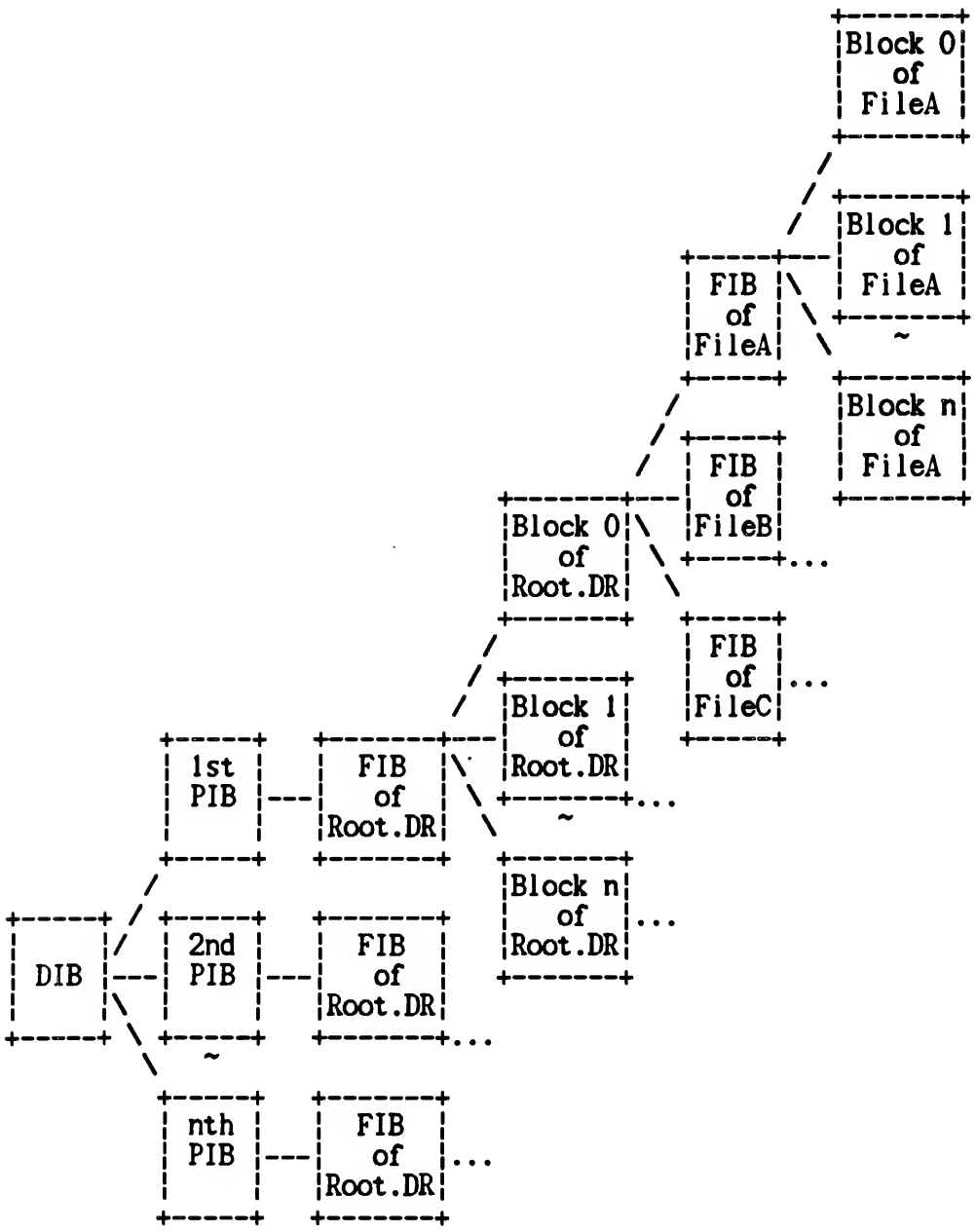
The FIB of the root directory, as well as the FIB of all other directories, contains pointers to each of the blocks that form the directory.

Each block of a directory contains pointers to the FIBs of the files that comprise the directory.

The FIB of a file contains pointers to the blocks that form the file.

Figure 2-1 depicts the data area links of the logical structure. Note that directory entries are hash coded by file name. The hash function specifies the block number of the directory to contain the entry. Therefore, directory entries are not necessarily accurately represented in Figure 2-1. However, Figure 2-1 accurately depicts placement if you assume that the file names hash to the same values. Chapter 3 provides more specifics on directory entries and the hash function.

Figure 2-1
Logical links for on-disk structure



The sections that follow provide complete details on the logical structures.

2.2 DEVICE INFORMATION BLOCK

The Device Information Block (DIB) identifies a device for use with the POS file system and serves as the ground zero entry point into the device's file structure. The DIB also contains a table which defines a mapping of 26 characters to 26 pairs of interpreter and system boot files. This table is read by the microprogram in the boot area. Module DiskIO.Pas defines the format of the DIB.

The DIB is the first logical block on a volume (LBN 0).

On a Shugart disk, the boot code occupies the first track. Thus, on a Shugart disk, LBN 0 is at cylinder 0, track 1, sector 0 (LDA 000000 140000, PDA 000400 000000). On a Micropolis disk, the boot code occupies the first two tracks. Thus, on a Micropolis disk, LBN 0 is at cylinder 0, track 2, sector 0 (LDA 000000 140000, PDA 001000 000000). On a 5.25" disk, LBN 0 is at cylinder 1, track 2, sector 0 (LDA 00000 190000, PDA 01000 00000). On a floppy disk, LBN 0 is at cylinder 5, track 0, sector 3 (LDA 000000 160000, PDA 000003 000005).

The DIB contains pointers to each Partition Information Block (PIB).

The following sections provide a detailed description of the DIB. Note that the DIB and all Partition Information Blocks (PIBs) use the same format; some words in the structure are relevant only in a DIB structure and some only in a PIB structure.

Section 2.2.14 provides a graphic representation of the DIB.

2.2.1 FreeHead

This double word contains the LDA for the start of a partition's free block list and is therefore not relevant to the DIB. The word values are 0 to indicate a nil pointer. Section 2.3 describes the significance of this double word.

2.2.2 FreeTail

This double word contains the LDA for the end of a partition's free block list and is therefore not relevant to the DIB. The word values are 0 to indicate a nil pointer. Section 2.3 describes the significance of this double word.

2.2.3 NumberFree

This double word contains an integer value that specifies the number of free blocks in a partition and is therefore not relevant

to the DIB. The word values are 0. Section 2.3 describes the significance of this double word.

2.2.4 RootDirectoryID

This double word contains the LDA for a partition's root directory and is therefore not relevant to the DIB. The word values are 0 to indicate a nil pointer. Section 2.3 describes the significance of this double word.

2.2.5 BadSegmentID

This double word contains the LDA for a partition's bad segment and is therefore not relevant to the DIB. The word values are 0 to indicate a nil pointer. Section 2.3 describes the significance of this double word.

2.2.6 BootTable

This 52-word array contains physical addresses for Qcode boots. Each double word of the 52-word array corresponds, successively, to the lowercase letters a through z for each .Boot file on a hard disk and to the uppercase letters A through Z for each .Boot file on a floppy disk. For example, if a system contains an a, C, and d boot, the first two words (words 0 and 1) of this array contain the physical address of the operating system to boot when lowercase a (the hard disk) is booted, the fourth and fifth words (words 3 and 4) contain the physical address of the operating system to boot when uppercase C (the floppy disk) is booted, and the sixth and seventh words (words 5 and 6) contain the physical address of the operating system to boot when lowercase d is booted.

2.2.7 InterpreterTable

This 52-word array contains physical addresses for microcode boots. Each double word of the 52-word array corresponds, successively, to the letters for each .MBoot file (MBoot files contain the Qcode interpreter microcode).

2.2.8 PartitionName

This quad word contains the name (one to eight characters) of the partition. In the DIB, the partition name is interpreted as the name of the device (for example, SYS). The name from this quad word is the default for the device portion of a file specification.

2.2.9 PartitionStart

This double word contains the LDA of the start of the partition. In the DIB, the partition start is interpreted as the first logical block (LBN 0) on the device. This double word always points to LBN 0 (hard disk LDA 000000 140000, floppy disk LDA 000000 160000).

2.2.10 PartitionEnd

This double word contains the LDA of the end of the partition. In the DIB, the partition end is interpreted as the last logical block on the device. This double word always points to the highest block number of the device.

2.2.11 SubPartitions

This 128-word array contains the LDA of the Partition Information Block (PIB) for each partition (thus, an array of 64 LDAs, one for each possible partition). As you create each partition, the Partition program (see Section 5.1) writes the address of its first block successively in this arrray.

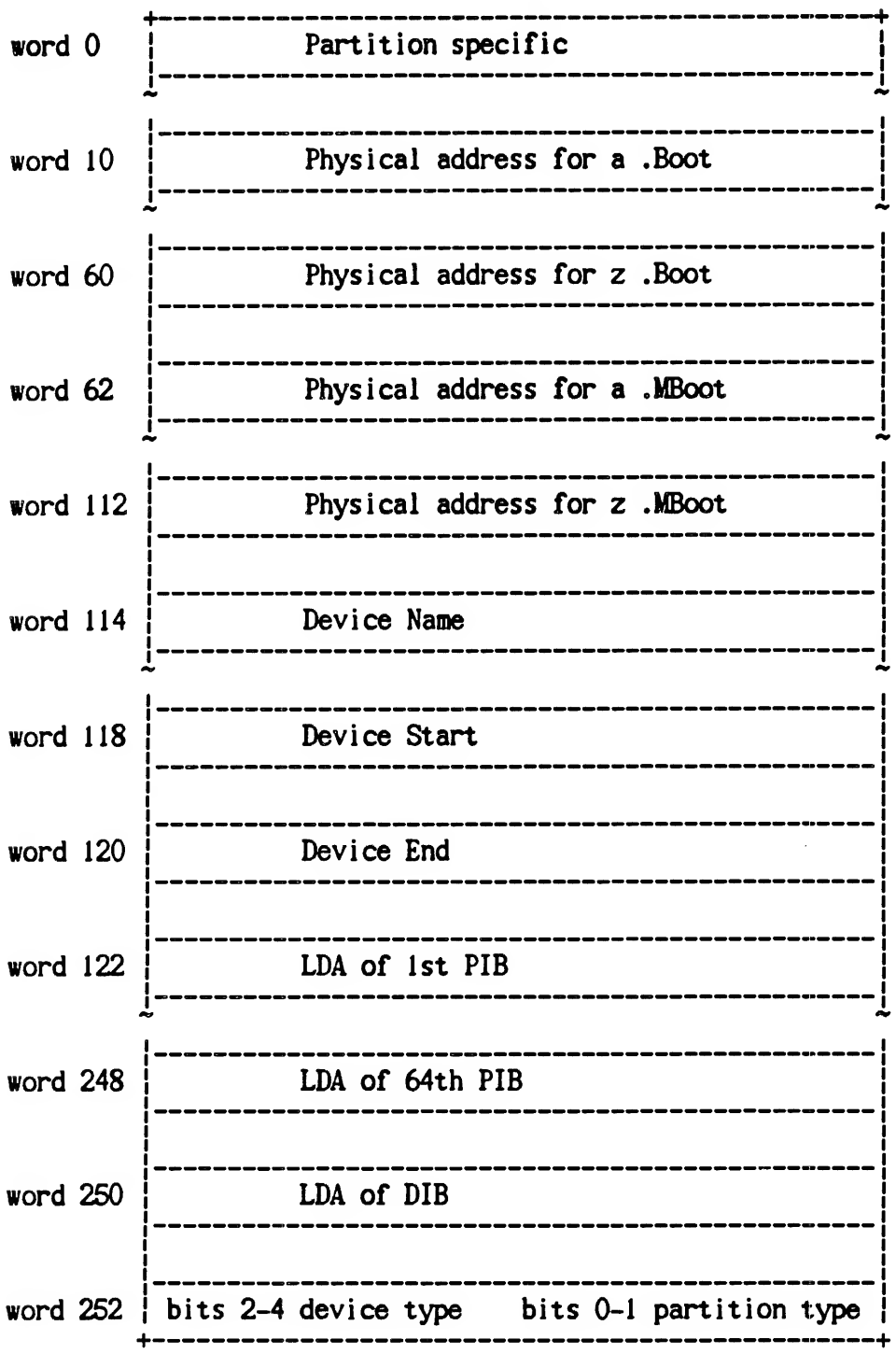
2.2.12 PartitionRoot

This double word contains the LDA of the root partition (that is, the DIB) and always points to LBN 0.

2.2.13 PartitionType/DeviceType

This word specifies partition type (root, unused, or leaf) and device type (Winchester 12-MB, Winchester 24-MB, floppy single density, floppy double density, unused1, unused2). In a DIB structure, the partition type is always root. The Partition program (see Section 5.1) writes the device type in this word when you partition the volume.

2.2.14 Disk Information Block Layout



words 253, 254, and 255 are unused

2.3 PARTITION INFORMATION BLOCK

A partition is a physically contiguous area of logical blocks. The fundamental operations on a partition are allocation and deallocation of blocks from and to a pool of blocks in the partition. The pool of blocks is known as the partition's free block list. You allocate blocks to form segments, which can in turn form files, and deallocate blocks to destroy segments. Since segments can form files which are organized by a directory structured name space, a partition also provides a root directory for all files allocated in the partition.

Module AllocDisk.Pas manages a partition's free block list as a doubly linked list of blocks. The PIB contains two pointers (Logical Disk Addresses) to the head (first) and tail (last) blocks on the free list. When a partition is mounted (module AllocDisk.Pas also provides these functions), its PIB is read into a table, PartTable, in memory. As blocks are allocated or deallocated, AllocDisk.Pas updates information in the PartTable. The only dynamically updated information in the PIB is that which describes the partition's free block list. This information is updated in PartTable after every block allocation or deallocation, but is only updated intermittently in the copy of the PIB on disk. Consequently, if the system terminates abnormally, the disk copy may be incorrect. If the pointer to the head of the free list is incorrect, the system finds the actual start by following the filler word in the logical header.

The first block of a partition contains the name and limits of the partition, information describing the free block pool of the partition, and the Logical Disk Address of the partition's root directory (Root.DR). This block is known as the Partition Information Block or PIB. The PIB replicates the DIB structure, as described in Section 2.2. However, the structure for the DIB defines the file system entry point for the device while the PIB structure defines the physically contiguous area of disk addresses for each partition. Thus, one PIB exists for each partition on the volume while exactly one DIB exists for the volume. Module DiskIO.Pas defines the format of the DIB/PIB structure.

Partitions always start on cylinder boundaries. The Partition program, see Section 5.1, writes the PIB in the first block of a partition. Therefore, a PIB always starts on a cylinder boundary. The LBN of the first PIB depends on the volume.

On a hard disk, the first PIB is at physical address cylinder 1, track 0, sector 0. For a 24-MB Shugart disk, the first PIB is LBN 210 (LDA 151000 140000). For a 12-MB Shugart disk, the first PIB is LBN 90 (LDA 055000 140000). For a 35-MB Micropolis disk the first PIB is LBN 72 (LDA 044000 140000). On a floppy disk, the first PIB is at physical address cylinder 6, track 0, sector 1 (LBN 6).

The location of the PIBs for subsequent partitions depends on the number of blocks you allocate to each partition. A typical allocation for hard disks is 10080 blocks per partition. This permits the Scavenger (see Section 5.2) to handle the partition in a single pass.

Each PIB contains a pointer to the partition's root directory File Information Block (FIB). The file Root.DR contains Logical Disk Addresses of the FIBs for files, including directories. Since directories are hierarchically structured, the root directory effectively points to all files within a partition.

The following sections provide a detailed description of the PIB. Note that the DIB and all PIBs use the same format; some words in the PIB structure are not relevant in a DIB structure.

Section 2.3.14 provides a graphic representation of the PIB.

2.3.1 FreeHead

This double word contains the LDA for the start of the partition's free block list. The FreeHead entry points to a block within the partition. This block is the actual start of the free list when the serial number and the previous pointer in its header are zero.

If the serial number and the previous pointer are not zero, the actual start of the free list is found by following the filler word until both are zero.

2.3.2 FreeTail

This double word contains the LDA for the end of the partition's free block list. Like the FreeHead entry, see Section 2.3.1, the FreeTail pointer in the PIB may not be accurate.

The FreeTail entry points to a block within the partition. This block is the actual end of the free list when the next pointer in its header is zero (zero indicates a nil pointer). If not, the actual end of the free list is found by following the next address pointers to zero.

2.3.3 NumberFree

This double word contains an integer value that suggests the number of free blocks in a partition. This value may not be accurate for the same reason that the accuracy of the FreeHead and FreeTail pointers is not guaranteed.

2.3.4 RootDirectoryID

This double word contains the LDA for the partition's root directory (Root.DR) File Information Block (FIB). Section 2.4 describes the FIB. The root directory is a file that contains other directories and/or files, refer to Chapter 1.

2.3.5 BadSegmentID

Each partition has a linked list of bad blocks. The bad blocks are withheld from the free list and are thus not used by the file system. This double word contains the LDA for a partition's bad segment.

2.3.6 BootTable

This 52-word array contains physical addresses for Qcode boots and is therefore not relevant to the PIB. The word values are zero to indicate nil pointers. Section 2.2 describes the significance of these words.

2.3.7 InterpreterTable

This 52-word array contains physical addresses for microcode boots and is therefore not relevant to the PIB. The word values are zero to indicate nil pointers. Section 2.2 describes the significance of these words.

2.3.8 PartitionName

This quad word array contains the name (up to eight characters) of the partition. You specify this name as input to the Partition program. The partition name can be the same as the device name, but each partition must have a distinct name.

2.3.9 PartitionStart

This double word contains the LDA of the first block of the partition. Since the PIB is the first block of a partition, the entry points to the partition's PIB.

2.3.10 PartitionEnd

This double word contains the LDA of the last block of the partition.

The entries for PartStart (see section 2.3.9) and PartEnd specify the size of each partition as physically contiguous disk addresses. You enter the number of blocks, and thus size, for each partition as input to the Partition program. The maximum size of a partition is 32768 blocks.

2.3.11 SubPartitions

This 128-word array contains the LDA of the PIBs for each partition and is therefore not relevant to any PIB itself. Section 2.2 describes the significance of these words.

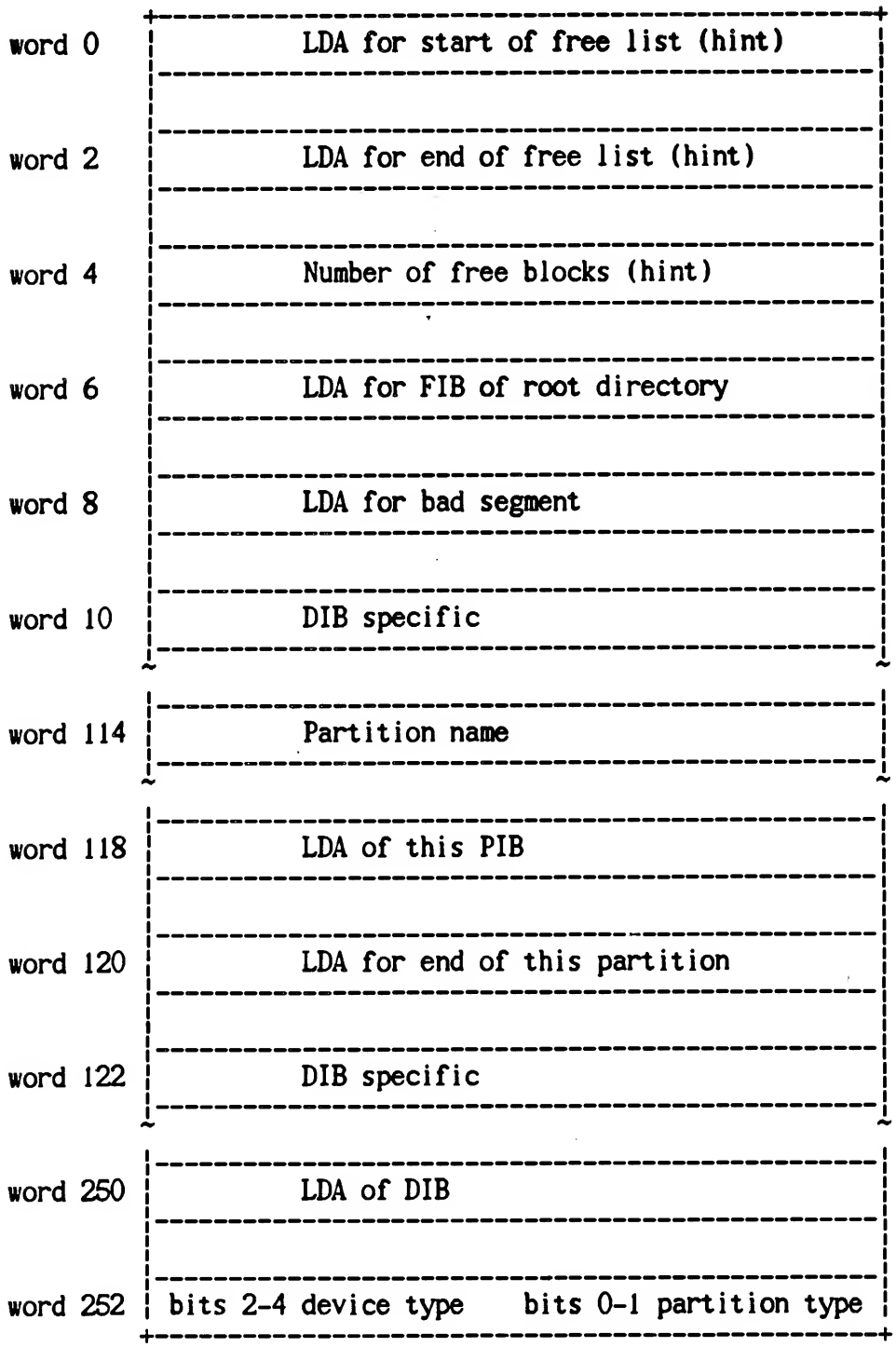
2.3.12 PartitionRoot

This double word contains the LDA of the root partition or DIB (LBN 0).

2.3.13 PartitionType/Device Type

This word specifies partition type (root, unused, or leaf) and device type (Winchester 12-MB, Winchester 24-MB, floppy single density, floppy double density, unused1, unused2). In a PIB structure, the partition type is always leaf.

2.3.14 Partition Information Block Layout



words 253, 254, and 255 are unused

2.4 FILE INFORMATION BLOCK

A single block of a segment or a file (including directories) describes each segment or file on a POS file structured volume and is referred to as a File Information Block (FIB). The FIB is relative logical block number -1. The blocks that form the usable part of a segment or file have relative logical block numbers ranging from 0 through 32767. The FIB identifies to which particular segment or file a specific block belongs; the LDA of the FIB is the serial number for all the blocks of a segment or file.

The FIB contains all the information necessary to access an individual segment or file as well as pointers to the blocks that form the segment or file. (Actually, the FIB only allows access to a segment's blocks; file access requires the DIB, PIB, and directory structures.)

Some entries in the FIB are not relevant when the structure describes only a segment. Specifically, the FIB of a segment has zero values for the first 52 words, while the FIB of a file has relevant values for these words.

Module DiskIO.Pas defines the format of the FIB structure.

The following sections provide a detailed description of the FIB. Section 2.4.9 supplies a graphic representation of the FIB.

2.4.1 FileSystemData

This 52-word packed record contains basic file system information, including the file name. The file system uses this record to access and maintain a file. The record is not used by the segment system. Module FileDefs.Pas provides the Pascal type definition for this packed record.

The first word (word 0) contains the number of blocks allocated to the file.

The second word (word 1) specifies the number of bits in the last block of the file, whether or not the file can be sparse, and whether the file is open for read, write, execute, or not open. The low order 12 bits (bits 0 through 11) specify the number of bits in the last block of the file. For example, when you allocate a block from the free list for use as the last block of a file and do not fill the entire block with data, the remainder of the block may contain random data or garbage. These 12 bits signal the end of significant data. Bit 12 specifies whether or not the file can be sparse. Bits 13 through 15 indicate the following: file not open; file open for read; file open for write; and file open for execute.

The third and fourth word (words 2 and 3) contain the date and time

at which the file was created. The time is expressed in the standard internal time stamp format, which has numeric fields for year, month, day, hour, minute, and second.

Words four and five contain the date and time, also expressed in the standard time stamp format, at which the file was last deaccessed after being accessed for a write operation.

The seventh and eighth words (words 6 and 7) contain the date and time of the last file access.

Word eight contains the file type. Any 16-bit value can be read. POS uses the following file types:

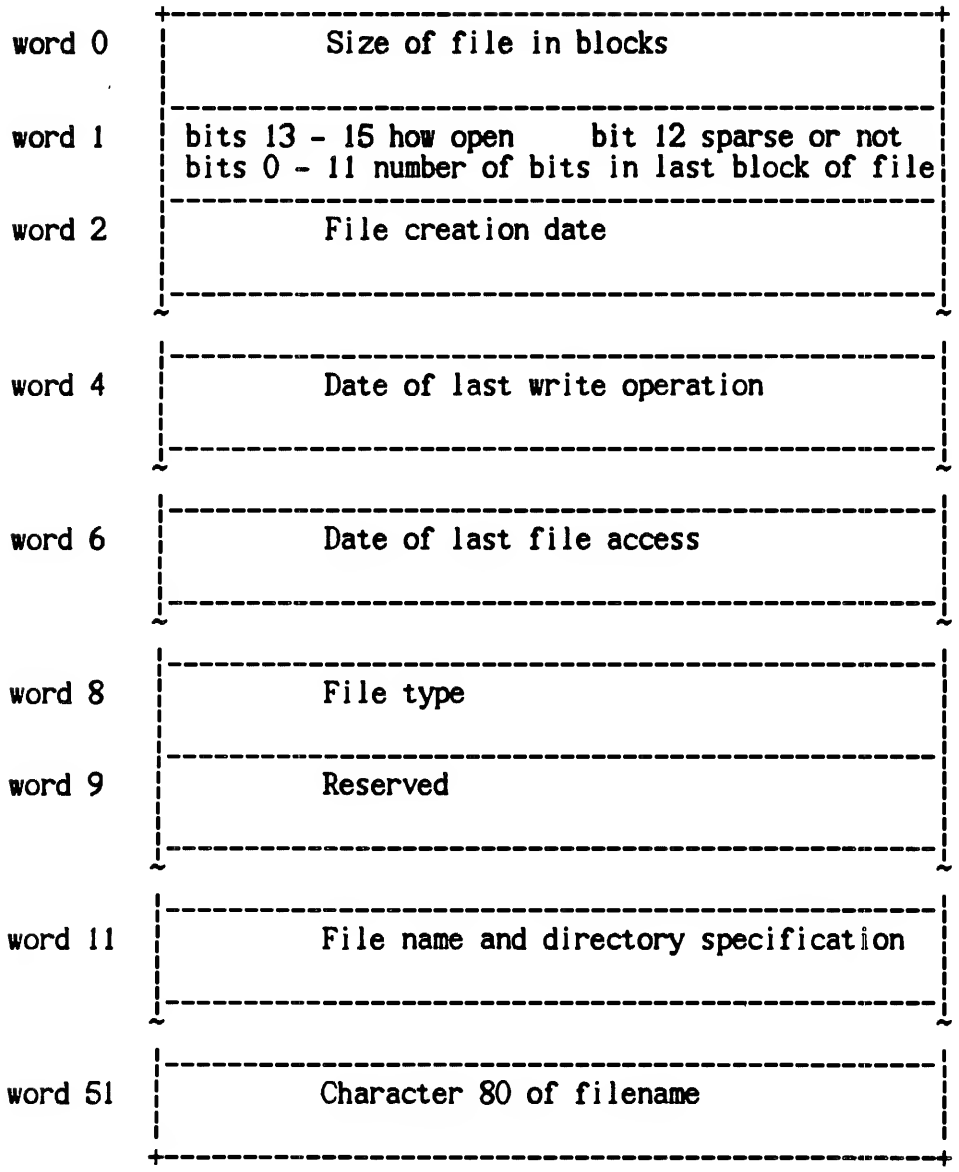
```
UnknownFile
SegFile
PasFile
DirFile
ExDirFile
FontFile
RunFile
TextFile
CursorFile
BinaryFile
BinFile
MicroFile
ComFile
RelFile
IncludeFile
SBootFile
MBootFile
SwapFile
BadFile
```

Module FileTypes.Pas defines the above file types.

The tenth and eleventh words (words 9 and 10) are not used, but are reserved for file protection.

Words 11 through 51 contain the partial file name (includes all directories, omits the partition and device specification). Partition and device specifications are not required since files cannot cross partition boundaries; the file must be in the same partition as its FIB, the FIB in the same partition as the PIB, and the PIB in the device's DIB. Note that the file name is a string; the low byte of the first word of a string always specifies the length of the string. Therefore, a file name with its associated directory specifications cannot exceed 80 characters.

The following is a graphical representation of the file system data portion of the FIB.



2.4.2 Random Index

The random index provides a list of the disk addresses of the blocks that form the segment or file. It consists of three parts: the direct index; the indirect index; and the double indirect index.

The direct index is contained within the FIB itself (FIB words 52 through 179) and references logical disk addresses of the first 64 blocks of a segment or file (relative logical blocks 0 through 63).

The indirect index is also contained within the FIB (FIB words 180

through 243), but the entries do not point to blocks within the segment or file. Rather, the indirect index entries point to 32 blocks that each contain 128 disk addresses of blocks in the segment or file. Thus, these entries form a one-level addressing scheme. The indirect index can address 4096 blocks of a segment or file (relative logical blocks 64 through 4160, since the direct index addresses 0 through 63).

Like the indirect index, the double indirect index is contained within the FIB (FIB words 244 through 247). The indirect index points to blocks which point to blocks in the segment or file. The double indirect index entries contain disk addresses of two blocks. Each of these blocks point to 128 other blocks that each contain 128 disk addresses of blocks in the segment or file. Thus, these entries form a two-level addressing scheme. In theory, the double indirect index can address 32768 blocks of a segment or file (relative logical blocks 4161 through 36929). However, 32767 is the maximum size of a partition. Furthermore, an overhead of 291 blocks is required to address the theoretical maximum. Therefore, the maximum size of a single segment or file in a partition is governed by the size limits of the partition and the number of overhead blocks required to address the blocks of the segment or file.

The blocks in the random index have negative file logical block numbers as follows:

Direct index: block -1 (within the FIB)

Points to relative logical blocks 0 through 63.

Indirect index: blocks -4 through -35

Points to relative logical blocks 64 through 4160

Double indirect index: blocks -2 and -3

block -2 points to blocks -36 through -163 (blocks -36 through -163 point to
relative logical blocks 4161 through 20545)

block -3 point to blocks -164 through -257 (blocks -164 through -257 point to
relative logical blocks 20546 through 32767, the theoretical maximum)

2.4.3 SegmentKind

This word specifies whether a segment is permanent, temporary, or bad. A permanent segment persists until explicitly destroyed. Temporary segments (for example, segments used for swapping) exist

only while the process that creates the temporary segment exists. Bad segments are malformed segments and are not readable.

2.4.4 Number of Blocks In Use

Since a file can be sparse (block n may exist when block n-1 has never been allocated), this word specifies the number of blocks that are actually in use by the file.

2.4.5 LastBlock

This word specifies the file relative logical block number of the largest block allocated to the file. Remember that since files can be sparse, this word does not specify the total number of blocks in use by the file, refer to Section 2.4.4.

2.4.6 LastAddress

This double word specifies the Logical Block Number of the last block allocated to the file.

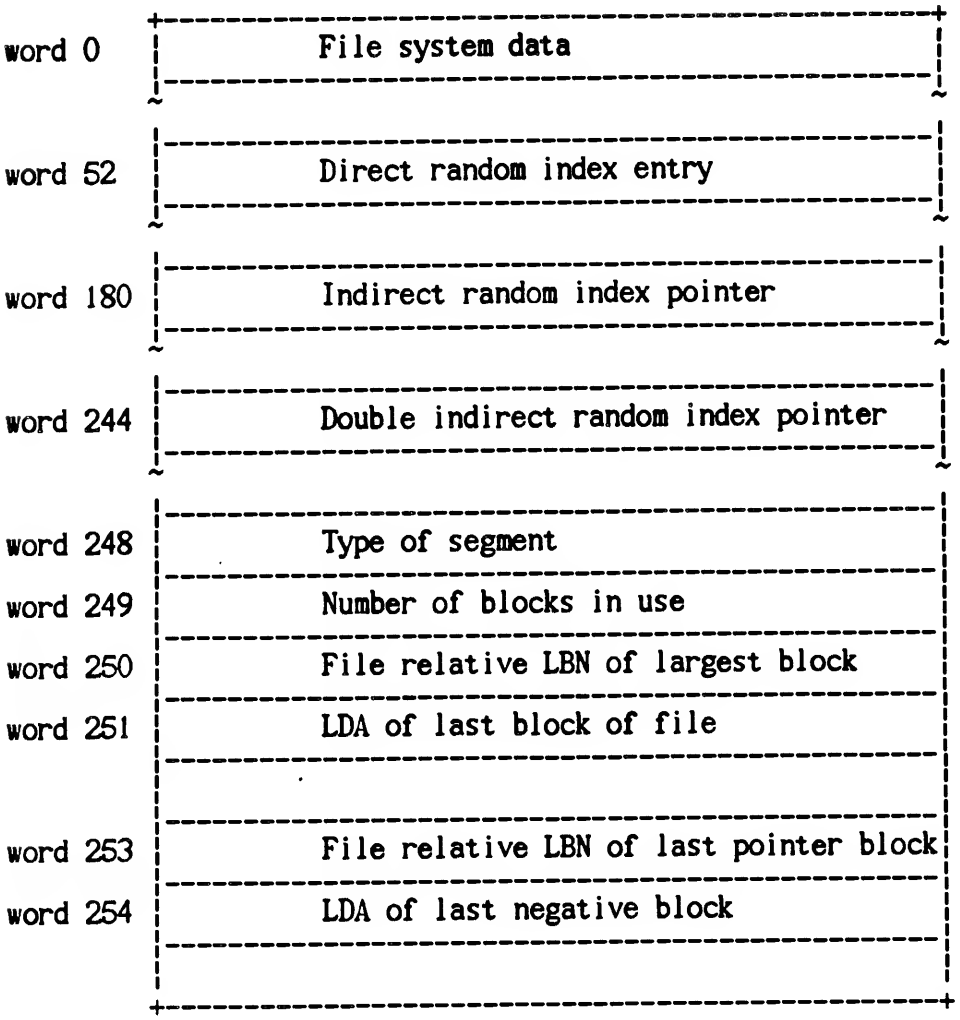
2.4.7 LastNegativeBlock

This word specifies the file relative logical block number of the file's pointer block with the largest absolute value (the smallest or most negative number).

2.4.8 Last Negative Address

This double word specifies the logical disk address of the last pointer block for the file.

2.4.9 File Information Block Layout



CHAPTER 3

FILE FORMATS

This chapter details the format of Segment (.Seg) and Directory (.DR) files.

3.1 SEG FILES

A segment file is produced when a program is compiled by the Pascal or FORTRAN compiler. The segment file is used primarily to hold the actual QCodes which the PERQ executes. In addition to the QCodes, the segment file includes information used by the consolidator, linker, loader and debugger. By default, segment files use .Seg as the extension. A collection of segment files with a run file contains all the information necessary for execution of a program. Pre-segment files are incomplete segment files produced by compilers that cannot resolve all routine references at compile time; they have the extension .Psg.

Segment files must conform to certain formatting conventions. A segment file consists of 512 byte blocks organized in seven groups: a header block; one or more code blocks; a routine dictionary; an import list; a routine name list; diagnostic information; and routine descriptor information.

The header block contains information about the size and contents of the module. The routine dictionary contains information necessary to the execution of routines. The import list contains module names and file names of imported modules. The routine name list contains names for the routines defined in the module. Diagnostic information is used for debugging, and routine descriptors are used to perform parameter checking.

3.1.1 Header Block

The first portion of the segment file contains 16 fields, as shown in Figure 3-1 (refer to Section 3.1.8 for precise sizes of the fields). The first field is a byte containing flags set by the compiler. Bit 0, the least significant bit, indicates whether the segment is a program or module. (A program is a special instance of a module, which includes a main body.) If ON, the segment is a program. Bit 1, if ON, indicates that the routine names (only) contained in the segment file are 14 characters long. If bit 1 is OFF, the names are 8 characters long. Bits 2 and 3 are defined for internal use, and the remaining 4 bits are reserved for future use. The high-order byte of this word contains the version number of the QCodes (the compiler generates the version number).

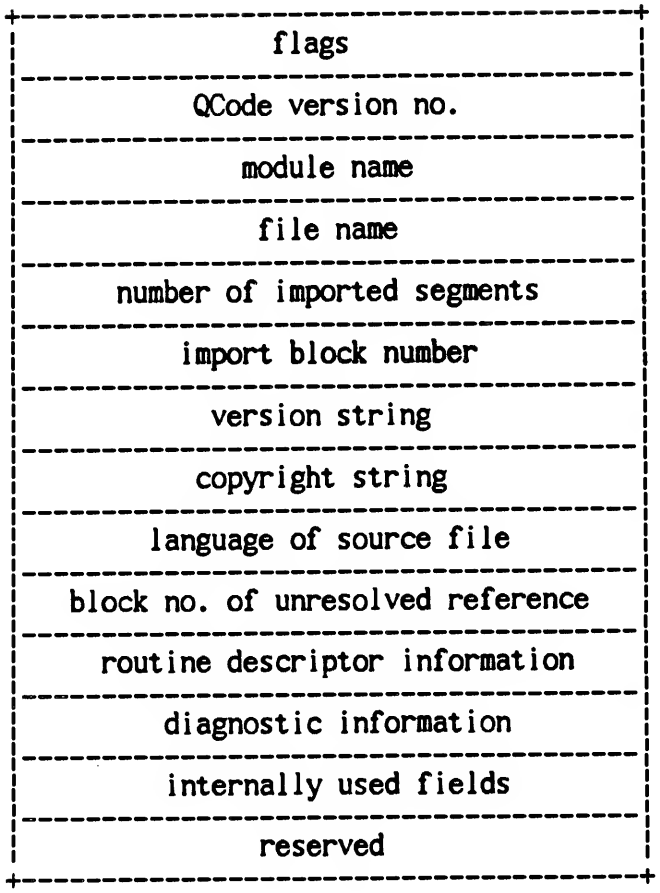
The next field (starting on the next word) contains the name of the module (as it appears in the source file). Module names are currently unique to 8 characters. The name of the source file from which the segment file was generated follows the module name. The filename string contains the full pathname of the source file from which the segment file was generated.

The fifth field gives the number of imported segments. This is followed by the import block number which is the number of the block containing the import list (described below) and the size of the global data block (referred to as the GDB in the QCode Reference Manual). (With no imported segments, the import block number is undefined.) The "version" and "copyright" fields follow. These strings may be specified using compiler switches or comments in the source file. The next field indicates the language of the source file. The next 3 fields are block numbers of the unresolved reference information (described in "Pre-segment Files" below), the routine descriptor information and the diagnostic information, respectively. If these blocks do not exist, these block numbers have the value zero. The last 3 fields are defined for internal use, and the remainder of the header block is reserved for future use.

The first two words of the second block (block 1) complete the header information. Word 0 points to the routine dictionary which follows the code. The value in word 0 is offset (in words) from the beginning of block 1 to the first word of the routine dictionary. Word 1 contains the number of routines within the segment.

Figure 3-1

Header Block 0

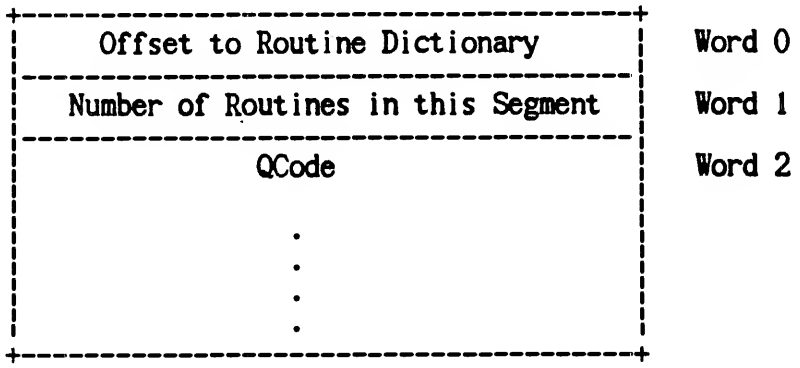


3.1.2 Code Blocks

The second entity is the code portion of the segment file (QCodes). The QCodes start with the third word (word 2) of the first block. The QCodes for each routine follow those of the previous routine on the next word boundary. The code section of the segment file may occupy several consecutive blocks.

Figure 3-2

Block 1



Block 1

The routine dictionary is aligned on the next quad-word boundary following the end of the QCodes. The dictionary contains an entry for each routine in the module and is padded to the end of the current block. A more complete description of the routine dictionary, as well as the format of an entry in the routine dictionary, may be found in the QCode Reference Manual.

3.1.3 Import List

Imports refer to segments which are external to the current module. The import list begins in the block following the routine dictionary (procedures, functions and exceptions in Pascal). The number of this block is given in the header block. The import list can take any number of blocks (including zero for no imports). The import list is ordered by segment number (the compiler-generated ISN [internal segment number]). Each entry in the import list consists of the 8 character module name followed by the name of the source file (for example, foo {from} foo.pas). Entries are word aligned and have constant length. Enough room is reserved for a 100 character file name (51 words). If a module entry needs less than the allotted space, it's padded with blanks.

3.1.4 Routine Name List

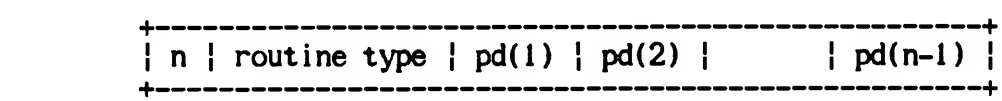
After the list of imported segments, the compiler provides the names of the routines defined in the module. The first routine name begins immediately after the last import entry. The entries in the routine name list are 8 or 14 characters long, are word-aligned, and are ordered by routine number (the order of their appearance in the source). Routine names may or may not fit entirely in the same block as the end of the import list.

3.1.5 Diagnostic Information

Diagnostic information begins in the block following the routine name list. This block number is found in the header block. It is used for program debugging. Currently, only segment files generated from FORTRAN sources contain diagnostic information.

3.1.6 Routine Descriptors

Routine descriptors begin in the block following diagnostic information. This block number is given in the header block. Currently, only segment files generated from FORTRAN sources contain routine descriptors. There is one descriptor for each routine defined in the module. They are word-aligned and are in the same order as the routine names. Each descriptor indicates whether the routine is a procedure (subroutine) or a function and describes each parameter. Routine descriptors allow type checking between the routines themselves and an external reference to the routine. A routine descriptor has the following format:



where

- each box above is a byte;
- n is one more than the number of parameters (the number of bytes that follow). The range for n is 1..255;
- routine type is 0 for procedures, st for functions where s (4 bits) is the size of the result and t (4 bits) is the result type;
- pd(i) is the parameter descriptor for parameter i.

Routine descriptors, if present, may occupy more than one block.

3.1.7 Pre-segment Files

Pre-segment files are produced by compilers that cannot resolve all routine references at compile time. They are incomplete versions of segment files and have the extension ".PSG". Currently, only the FORTRAN compiler produces pre-segment files and only in cases where it is not given enough information to resolve the references. The format and contents of a pre-segment file are identical to that of a segment file with the following exceptions:

- at least one call instruction in the QCodes section needs patched.
- additional information describing the unresolved routine references exists at the end of the file.

The unresolved reference information begins in the block following routine descriptors. This block number is given in the header block. The information is a sequence of reference descriptors of the following form:

where

- the call block number is the pre-segment file block number containing the call that needs resolved (1 word).
- the offset is the position of the call instruction within the block. It is 1 word and has a range of 0..511.
- the name of the routine is given to 8 or 14 characters, left-justified, padded with blanks if necessary.
- the routine descriptor gives information about the reference to the routine. See the section

on "Routine Descriptors" above. The one exception is that a routine descriptor inside of a reference descriptor can consist of one byte with the value 0 to signify a routine that is a parameter.

One reference descriptor exists for each unresolvable routine reference. If a routine is referenced multiple times, multiple reference descriptors for that routine will exist. The routine descriptor of the reference can be compared with the descriptor (in the routine descriptor section) of the file containing the routine. Reference descriptors are word aligned. The first reference descriptor begins on the first word of the first unresolved reference information block. The zeroth word contains the number of reference descriptors that follow. Unresolved reference information in pre-segment files may occupy more than one block.

3.1.8 Field Definitions

Users who need to manipulate segment files are encouraged to import the declarations found in the file "Code.Pas". The necessary definitions to access portions of segment and run files may be found in this file. Pertinent sections are included here which give the actual definitions and lengths for fields described above.

```
const
    QCodeVersion = 4;     { Current QCode Version Number }
    FileLength   = 100;   { Max chars in a file name }
    SegLength    = 8;     { Max chars in a segment file name }
    CommentLen   = 80;    { Length of comment and version
                            strings in segment files }
type
    SNArray = packed array[1..SegLength] of Char;  { Names in
       segment files }

    pFNString = ^FNString;

    FNString = String[FileLength];  { File name }

    QVerRange = 0..255;      { Range of QCode version numbers }

    { Segment file: }

    Language = (Pascal, Fortran, Imp);

    pSegBlock = ^SegBlock;

    SegBlock = packed record case integer of
```

```
        { zeroth (header) block: }
        0:  (ProgramSegment: boolean; { 1 if program }
             LongIds       : boolean; { 1 if 14 character
                                        ids }
             DbgInfoExists : boolean;
             OptimizedCode : boolean;
             SegBlkFiller  : 0..15;   { reserved }
             QVersion      : QVerRange; { QCode version no.
                                          (high byte)}
             ModuleName    : SNArray;
             FileName      : FNString;
             NumSeg        : integer;  { no. of imports }
             ImportBlock   : integer;
             GDBSize       : integer;
             Version       : String[CommentLen];
             Copyright     : String[CommentLen];
             Source        : Language;
             PreLinkBlock  : integer;  { Unresolved
                                         references }
             RoutDescBlock : integer;  { Routine
                                         descriptors }
             DiagBlock     : integer); { Diagnostics }
             QMapFileName  : FNString;
             SymFileName   : FNString;
             CompId        : TimeStamp);
        { first block: }
        1:  (OffsetRD      : integer;
             RoutsThisSeg  : integer);
        2:  (Block: array[0..255] of integer)
      end;

  CImpInfo = record case boolean of   { Import List Info -
                                        as generated
                                        by the compiler }
               true: ( ModuleName: SNArray; { Module
                                              identifier }
                       FileName:  FNString { File name }
                     );
               false:( Ary: array [0..0] of integer)
             end;

  SegFileType = file of SegBlock;
```

3.2 DIRECTORY FILES

Since directories are simply files that can contain names and addresses of files, the files contained in a directory can be other directories. Thus, directories can be nested to form a hierarchical, multi-directory structure. You can construct directory hierarchies of arbitrary depth and complexity to structure files in whatever manner is convenient.

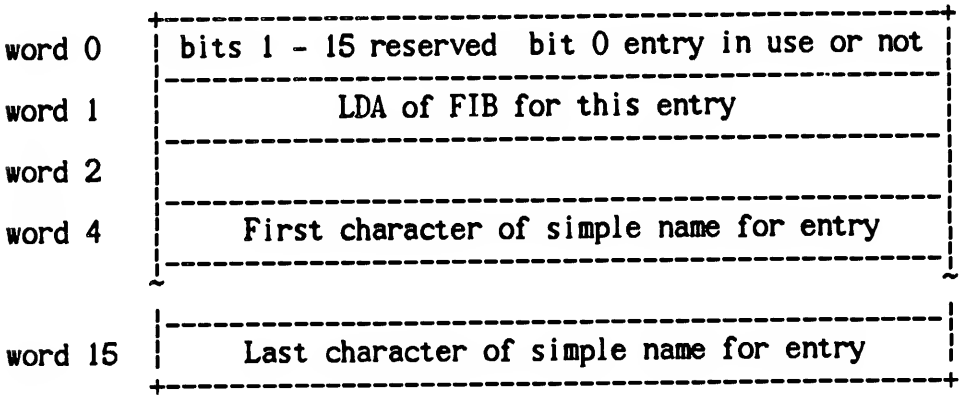
A directory entry is a 16-word packed record that points to the FIB of a file within the directory. Module DiskIO.Pas defines the format of a directory entry.

The first word of each directory entry specifies whether or not the entry is in use; bit 0 is set if the directory entry is valid. Bit 1 is reserved; in future systems, if the bit is set, the file associated with the entry has been deleted, but the disk blocks not yet returned to the free list. Bits 2 through 15 are reserved for future use. Note that reserved bits must be zero, but programs reading reserved bits should not assume that the bits are in fact zero.

The second and third words of a directory entry contain the logical disk address of the FIB of this entry.

The remaining 13 words contain the file name portion of a file specification. Since a directory can only contain entries for files within the partition, and thus device, where the directory itself is located, these words do not include the device, partition, and directory portions of a file specification. A file name that omits the device, partition, and directory portions is known as a SimpleName. Note that the file name is a string; the low byte of the first word of a string always specifies the length (in octal) of the string. Therefore, a simple file name cannot exceed 25 characters.

The following is a graphical representation of a directory entry.



Directory entries are hash coded by file name to decrease the lookup time. The hash function specifies the block of the directory in which the file name and the address of the file's FIB are written. When a block of the directory is full (it contains 16 directory entries), successive or overflow directroy entries destined for that block are written in block n + 31.

Since the hash function specifies the block of the directory to contain the entry, directories often contain unallocated blocks between allocated blocks.

Files that contain unallocated blocks between allocated blocks are known as sparse files.

CHAPTER 4

FILE OPERATIONS

This chapter describes basic file system operations.

A file is a hierarchical structure consisting of logical blocks allocated from a partition's free block list, a segment with an information block (FIB), and an entry in a directory. The file system consists of several cooperating modules that manage the various elements of a file.

Some of the modules that comprise the file system simply supply constants or type definitions and are not directly involved in file operations. Examples of these modules are: FileDefs.Pas; FileTypes.Pas; and IO_Private.Pas. Other modules use the constants and type definitions to perform a file operation. Examples of these modules are: AllocDisk.Pas; FileAccess.Pas; FileSystem.Pas; and IO_Unit.Pas.

The file system modules also use some of the operating system modules (for example GetTimeStamp.Pas) to perform related functions.

The modules with a specific jurisdiction over file system operation (modules that allocate blocks from the free block list, implement IO to logical blocks, read, write, and create segments, and perform file operations) are:

AllocDisk.Pas	FileUtils.Pas
DiskDef.Pas	IODisk.Pas
DiskIO.Pas	IOFloppy.Pas
DiskUtility.Pas	IO_Unit.Pas
FileAccess.Pas	ReadDisk.Pas
FileDir.Pas	VolumeSystem.Pas
FileSystem.Pas	

AllocDisk.Pas presides over each partition's free block list; it contains procedures to allocate and deallocate blocks within a partition. AllocDisk.Pas also contains procedures to mount and

dismount devices and partitions and thus maintains DiskTable and PartTable.

DiskDef.Pas supplies variables, constants, and types to the other file system modules, defines the control structure for CIO and EIO disks, and describes the EIO disk microcode to Pascal interface.

DiskIO.Pas defines the format of the logical structures (DIB/PIB, FIB, directory entries, and directory blocks).

DiskUtility.Pas supplies procedures to the volume system.

FileAccess.Pas supports the concept of segments, distinct from the directory structure. The module allows you to create, read, write, shorten (truncate), and delete segments. Since a segment is the foundation of a file, most file operations make calls to the procedures and functions in FileAccess.Pas.

FileDir.Pas supports the directory structure for the file system; it provides procedures to write and delete directory entries and to lookup files by name. The module also implements the file hash function (the hash function specifies the block of the directory to which the directory entry is written).

FileSystem.Pas provides the high-level interface to file operations; it creates, reads, writes, and closes files.

FileUtils.Pas extends the basic services of the file system (as provided by module FileSystem.Pas); it allows you to delete and rename files, specify a searchlist, add items to or subtract items from the searchlist, and obtain file names from a directory. FileUtils.Pas also provides a function to create new directories.

IODisk.Pas implements low level operations to hard disk and includes routines to convert between physical and logical disk addresses and logical block numbers.

IOFloppy.Pas implements low level operations to floppy disks.

IO_Unit.Pas provides procedures to perform I/O on various devices. The procedure UnitIO provides the interface to hard and floppy disks (among other devices) and supports read or write access to any physical block on either disk. Its syntax is:

```
UnitIO(Unit        : UnitRng;
       DataBuf     : IOBufPtr;
       Command     : IOCommands;
       ByteCount   : integer;
       DiskAddress : double;
       HeaderBuf   : IOHeadPtr;
       StatusBuf   : IOStatPtr);
```

UnitRng specifies the disk to access. The constant 1 specifies the hard disk and 3 specifies the floppy disk.

DataBuf is a pointer to the caller's buffer for data.

IOCommands is an enumerated type that specifies commands for devices; the relevant values are IORead, IOWrite, and IOWriteFirst. IORead and IOWrite specify a read or a write of the data block on the disk to or from the buffer at DiskAddress. IOWriteFirst specifies a write of both the data block and the logical header block from the buffers. The hard disk enforces a degree of checking of the contents of logical headers. It requires that the first 6 bytes of the logical header on the disk match a data structure supplied by the caller in the IO control block. UnitIO reflects this checking by passing the data in the buffer at HeaderBuf to the StartIO level for checking when Command is IORead or IOWrite.

The integer for ByteCount must be a multiple of 128 if Unit = Floppy (3) and a multiple of 512 if Unit = HardDisk (1).

DiskAddress is an encoding of the three components of a physical disk address (cylinder, track, and sector). The actual encoding is a function of the device; for floppy DiskAddress[0] holds the sector number and DiskAddress[1] holds track * FlpCyls + cylinder. For the hard disk DiskAddress[1] is ignored and DiskAddress[0] is partitioned bitwise into three fields: cylinder (8 bits), head (3 bits), and sector (5 bits).

HeaderBuf is a pointer to the caller's buffer for header data; this only applies to the hard disk.

StatusBuf is a pointer to a structure of the caller that reports errors.

ReadDisk.Pas provides a buffer system for read and write operations to the hard and floppy disks.

VolumeSystem.Pas provides uniform abstractions of the disks. This module makes a disk appear as an array of pairs of data blocks and logical headers. VolumeSystem.Pas defines a general address Type with two components, one to specify a disk and another to specify an index into the array on that disk. The module also assists in mount and dismount operations (mounting a disk means reading a symbolic name from a known address on that disk and recording a mapping of that symolic name to an identifier for the disk) and provides operations to determine the number of pages (pairs of data blocks and logical headers) and the last valid address on a given disk.

The sections that follow present a step-wise refinement through each of the following file operations:

Create
Read
Write
Delete
Close

4.1 CREATING A FILE

Use the function FSEnter in module FileSystem.Pas to create a file. The function takes a file specification as a parameter and returns the FileID of the created file:

```
FSEnter(pathname): FileID;
```

If the device, partition, or directory portions of the file specification do not exist or if the directory or file name portions are longer than 25 characters, FSEnter raises an exception, FSBadName, and returns a FileID of zero. Note that you must specify at least a file name from 1 to 25 characters as a parameter; null file names raise the FSBadName exception.

To create a file, FSEnter first ensures a complete file specification parameter and then determines whether or not the file already exists. For an existing file, FSEnter simply changes the file's write and access dates in the FIB and returns the FileID of the existing file. For a new file, FSEnter allocates a block from the partition's free block list to construct an FIB for the file, makes an appropriate directory entry, and then returns the FileID of the new file.

The procedure FixFileName (in module FileSystem.Pas) ensures a complete file specification. A complete file specification consists of four portions and takes the form:

```
device:partition>directory1>...directory9>filename
```

A complete file specification is referred to as a full filename; it specifies the complete path for a file. Since the file system applies defaults for device, partition, and directories, users need not specify a full filename. A file specification that omits the device and partition portions (includes only directories and filename) is referred to as a partial file specification. A file specification that omits all portions except the file's name is referred to as a simple file specification.

Procedure FixFileName adds defaults, as necessary, to a partial or simple file specification. For example,

```
mydir>myfile.txt
```

is valid as the pathname parameter in the call to FSEnter, but the parameter omits the device and partition portions of a file specification.

Thus, FixFileName would insert the device name (for example, SYS) and the partition name (for example, BOOT) in the partial file specification above to yield SYS:BOOT>MYDIR>MYFILE.TXT.

The function GetFileID (in module FileDir.Pas) determines whether or not a file exists; it returns the Logical Disk Address of the information block of a file (the SegID). If the file does not exist, both words are zero.

For an existing file, procedure ReadSpiceSegment (in module FileAccess.Pas) reads the information block into a buffer. Procedure WriteSpiceSegment (also in module FileAccess.Pas) writes new values for the file write and access dates in the buffer representing the information block and then writes the buffer back to disk. Words 4-5 (FileWriteDate) and 6-7 (FileAccessDate) of the file's FIB now reflect the operation.

For a new file, function CreateSpiceSegment (in module FileAccess.Pas) allocates a block from the free block list and writes the information block of a segment to represent the new file. CreateSpiceSegment calls the function AllocDisk (in module AllocDisk.Pas) to obtain a block.

Function AllocDisk allocates a free block from a partition's free block list and returns the Logical Disk Address of the newly freed block.

The free block pool must always contain at least one block. Therefore, if the block to allocate is the last free block in the partition (that is, both word values for the next address pointer in the block's logical header are zero), function AllocDisk raises an exception, PartFull. If the free list is inconsistent (that is, when the previous address pointer in the logical header of the second free block doesn't point to the first block of the free list) function AllocDisk raises an exception, BadPart. In this case, you must run the Scavenger to reconstruct the free block list.

Whenever you allocate a logical block from the free list, AllocDisk updates the values in the memory structure PartTable (for the head of the free list and the number of free blocks) and writes the logical header of the new head of the free list (zeroes next and previous pointers and sets filler to next free block).

CreateSpiceSegment uses the Logical Disk Address from function

AllocDisk to write the information block. The logical header of the information block is written first (serial number is LDA from AllocDisk, logical block is -1 to indicate the FIB, next and previous pointers zero, and filler pointing to next free block) followed by FIB words 52 through 255. The values in the logical header area of the FIB are:

Serial Number is LDA from AllocDisk
Logical Block Number is -1 to indicate FIB
Filler is next free block
Next Address is 000000 000000
Previous Address is 000000 000000

The values in the data area of the FIB reflect a segment as follows:

File System Data (words 0 - 51) is 0 (not written)
Random Index (words 52 - 247) is 000000 000000
Type of Segment (word 248) is permanent
Number of blocks in use (word 249) is 1
LBN of largest block (word 250) is -1
LDA of last block (words 251 - 252) is LDA from AllocDisk
LBN of last pointer block (word 253) is -1
LDA of last negative block (words 254 - 255) is LDA from AllocDisk

At this point, the file lacks a directory entry and its FIB omits file system data (FIB words 0 through 51 have zero values).

Function PutFileID (in module FileDir.Pas) writes the directory entry for the new file. To write the directory entry, PutFileID first finds the partition's root directory. It then follows the list of sub-directories to the target directory. When PutFileID locates the target, it executes the hash function to determine the relative logical block number of the target directory in which to write the file name and then writes the directory entry. Note that a directory entry is a "simple name" that includes only the file name portion of a file specification.

Procedures ReadSpiceSegment and WriteSpiceSegment update the file system data portion of the FIB. The values for FIB words 0 through 51 are as follows:

File size (word 0) is 0
Bits in last block (word 1) is 0
File create date (words 2 - 3) is timestamp of current time
Last write date (words 4 - 5) is timestamp of current time
Last access date (words 6 - 7) is timestamp of current time
File type (word 8) is 0 (unknown file)
Words 9 - 10 are unused
File name (words 11 - 51) is partial name (includes all directories, but omits device and partition portions of file

specification).

Note that since only the FIB exists for the file, it is a zero length file (file size is 0).

Function SegIDToFileID in module FileSystem.Pas converts the SegID of the newly created file into a FileID. FSEnter returns this value. You can use procedure FSBlkWrite (in module FileSystem.Pas) to add blocks to the file as described in Section 4.2.

4.2 WRITING A BLOCK IN A FILE

Use the function FSBlkWrite in module FileSystem.Pas to add a block to a file or to update an existing block of a file. The function takes three parameters: the FileID of the file to write; the relative logical block number (starting at zero) of the block to write; and a buffer to hold the data to write in the block of the file.

```
FSBlkWrite(UserFile:FileID; Block:BlkNumbers;
           Buff:PDirBlk);
```

Note that the parameter for the file to write is a FileID.

Typically, unless you are adding a block to a newly created file (function FSEnter returns a FileID), you refer to files by file name and not by FileID. To write to a file by file name, first convert the file name to a FileID.

Module FileSystem.Pas includes two functions, FSLocalLookUp and FSLookUp, that convert a file name to a FileID.

Both functions take the same parameters.

```
FSLocalLookUp
    or      (pathname; Var BlkInFile,BitsInBlk):FileID;
FSLookUp
```

The pathname parameter is the file specification to search for. If the device, partition, directory, or file name portions of the file specification do not exist or if the file name portion is longer than 25 characters, the functions raise an exception, FSNotFnd, and return a FileID of zero. If the file is found, the functions write an integer value for the variable parameters "blocks in file" and "bits in last block" (from FIB words 0 and 1), update the file access date (FIB word 6), and return the FileID.

The difference between the two functions is that FSLocalLookUp ignores the searchlist and performs the lookup operation only in the current path. FSLookUp performs the lookup operation using the paths on the searchlist.

With a valid FileID, you can write blocks of a file. To write (or update) a block of a file, FSBlkWrite converts the FileID to a SegID and then either updates an existing block or writes a new block.

The function FileIDToSegID in module FileSystem.Pas converts the one-word FileID of the file to write into a two-word SegID.

Procedure WriteSpiceSegment in module FileAccess.Pas writes the new block or updates an existing block. To write a block, WriteSpiceSegment first ensures that the SegID is a valid File Information Block. (Functions SegAddr and CheckHeader, both in module FileAccess.Pas, perform this test.) The FIB is valid when its logical header's serial number matches the specified SegID and its LBN is negative one (-1). An invalid FIB raises an exception, NotAFile.

If the FIB is valid, WriteSpiceSegment determines whether or not the specified relative logical block is allocated to the file. Function ReadHeader (in module ReadDisk.Pas) reads the header of the relative logical block to write.

If the Logical Block Number in the header matches the specified relative logical block to write (the block is allocated to the file), WriteSpiceSegment copies the data from the user buffer into a memory buffer representing the disk block. Following a flush operation, the new data is written from the memory buffer to the disk block. Note that you are free to re-use your buffer containing data to write.

If the Logical Block Number in the header is not allocated to the file, WriteSpiceSegment first allocates a block from the free block list. Function AllocDisk (in module AllocDisk.Pas) allocates free blocks; refer to Section 4.1. WriteSpiceSegment then writes the new block's logical header and updates the random index in the file's FIB to reflect the new block. Finally, the function copies the data from the user buffer into a memory buffer and then writes the memory buffer to the new disk block.

4.3 READING A FILE

Use the function FSBlkRead in module FileSystem.Pas to read a block of an existing file. The function takes three parameters: the FileID of the file to read; the relative logical block number (starting at zero) of the block to read; and a buffer to hold the data read from the block of the file.

```
FSBlkRead(UserFile:FileID; Block:BlkNumber; Buff:PDirBlk);
```

Note that the parameter for the file to read is a FileID and not a

file name. To refer to files by file name and not by FileID, first convert the file name to a FileID as described in section 4.2.

With a valid FileID, you can read a block of a file. FSBlkRead first converts the FileID to a SegID and then reads the block from the segment.

The function FileIDToSegID in module FileSystem.Pas converts the one-word FileID of the file to read into a two-word SegID.

Procedure ReadSpiceSegment in module FileAccess.Pas reads the block. To read a block, ReadSpiceSegment first ensures that the SegID is a valid File Information Block and that the specified block to read is actually allocated to the file. The FIB is valid when its logical header's serial number matches the specified SegID and its LBN is negative one (-1). An invalid FIB raises an exception, NotAFile. If the block to read is not part of the file, FSBlkRead zeroes the buffer you specified to hold the data.

If the FIB is valid and the block is allocated to the file, FSBlkRead copies the data from the block into the buffer.

4.4 DELETING A FILE

Use the procedure FSDelete in module FileUtils.Pas to delete a file. Since a file is a segment with a directory entry, FSDelete removes the directory entry that names the file and then destroys the segment that represents the file. The procedure takes the name of the file to delete as the only parameter.

```
FSDelete(filename : Pathname);
```

FSDelete ignores the searchlist and only deletes a file from the current path. Thus, to delete a file from other than the current path, you must enter a complete file specification. If the device, partition, or directory portions of the file specification do not exist or if the directory or file name portions are longer than 25 characters, FSDelete raises an exception, DelError, and exits. If you omit the file name altogether, FSDelete simply exits; null file names do not raise an exception.

To delete a file, FSDelete first ensures a complete file specification as described in Section 4.1. Note that FixFileName adds only the current device, partition, and directory portions to a partial or simple file specification.

The function DeleteFileID (in module FileDir.Pas) removes the directory entry for the file.

The procedure DestroySpiceSegment invalidates the File Information Block and returns the file's allocated blocks to the free list. To

invalidate the FIB, DestroySpiceSegment simply zeroes the serial number in the FIB's logical header. Procedure DeallocChain (in module AllocDisk.Pas) returns the blocks to the free list. To return the blocks, DeallocChain adds the file's most negative block (the last block of the random index) and the file's last logical block to the end of the free list. The procedure updates the most negative block's logical header as follows:

Serial number is 000000 000000 to indicate a free block
Logical Block Number is unchanged (updated when block is allocated)
Filler is LBN of free list head
Next LDA is unchanged (still points to next block of file)
Previous LDA is address of prior end of free list

The file's last logical block becomes the end of the free list; DeallocChain updates its header as follows:

Serial number is 000000 000000 to indicate a fre block
Logical Block Number is unchanged (updated when block is allocated)
Filler is LBN of free list head
Next LDA is 000000 000000 to indicate free list tail
Previous LDA is unchanged (still points to previous block of file)

DeallocChain then updates the PIB entries for FreeTail (PIB words 2 and 3) and NumberFree (PIB words 4 and 5) to include the additional free blocks. The number of blocks previously allocated to the file are added to the value for NumberFree.

Note that the deleted file's doubly linked list of blocks (next and previous LDAs) remains intact; the logical headers of these blocks are updated as the blocks are allocated.

4.5 CLOSING A FILE

Use the procedure FSClose in module FileSystem.Pas to close a file and assert file length. The procedure takes three parameters: the FileID of the file to close; the number of blocks in the file; and the number of bits in the last block.

```
FSClose(UserFile:FileID; Blcks,Bits:Integer);
```

Note that the parameter for the file to close is a FileID and not a file name. To refer to files by file name and not by FileID, first convert the file name to a FileID as described in section 4.2.

With a valid FileID, you can close a file. FSClose first converts the FileID to a SegID and then reads the FIB.

If FileType is a DirFile (the value in FIB word eight is 3), FSClose raises an exception, FSDirClose. You can continue from this exception and close a Directory file, but this is not recommended; you could render the directory unusable.

If the file to close is not a DirFile or if you continue from an FSDirClose exception, FSClose writes the values you specified in FIB word 0 (the number of blocks allocated to the file) and in bits 0 through 11 of FIB word 1 (the number of bits in the last block).

Finally, FSClose truncates the file so that the file contains the number of blocks you specify. The final truncate operation quarantees that the file contains only allocated blocks. This is why it is not recommended that you continue following a FSDirClose exception; directory files can be sparse and truncating a DirFile could remove inuse blocks of the directory.

CHAPTER 5

FILE SYSTEM UTILITIES

This chapter describes the PERQ file system utilities Partition, Scavenger, MakeBoot, and FixPart.

5.1 PARTITION PROGRAM

The Partition program creates and modifies partitions. Creating a partition usually destroys all old data in the physically contiguous disk area where the partition is made.

The file system restricts partitions on a device to fewer than 32768 logical blocks. However, to permit the Scavenger, described in Section 5.2, to handle the partition in one pass, each partition should have 10080 or fewer blocks. All partitions start and end on cylinder boundaries; the Partition program enforces this constraint.

To run the Partition program, simply type:

```
Partition
```

in response to the default PERQ prompt. The program then asks a series of questions. Your responses to the questions specify parameters such as device and partition names and partition size. The following lists all of the possible questions. Explanatory text follows each question. Note that your responses control the logical flow of the questions; depending on your responses, not all the questions appear.

1. Do you want to debug? (does not do any writes) [No]

   The response to this question specifies whether or not the Partition program actually initializes or modifies a device.

   If you respond Yes, the program simply continues with the question sequence, but the device is not modified. Respond Yes to familiarize yourself with the logical flow of Partition questions and to prepare your responses.

   If you respond No, the program modifies the device based on your responses to succeeding queries.

   The default response is No (the program modifies the device).

2. Partition HardDisk (H) or Floppy (F)?

Your response to this question specifies the device to modify. The choices are the Hard disk or the Floppy disk.

If you respond H (to indicate the hard disk), Partition checks to see what type of disk drive you are using and asks whether your disk is a MICROPOLIS 8-inch drive, a Shugart 14-inch drive, or a 5.25-inch drive. See question 3 if you are running on a PERQ2, question 4 if running on a PERQ, and question 5 if using a 5.25-inch disk.

If you respond F (to indicate the floppy disk), Partition asks whether the floppy is single or double sided. See question 9.

3. This seems to be a MICROPOLIS 8-inch disk.
Is this right? [Yes]

This question confirms the disk choice.

The next question is number 10.

4. This seems to be a SHUGART 14-inch disk.
Is this right? [Yes]

This question confirms the disk choice. Partition then checks to see whether the disk is a 12- or 24-megabyte disk. It then asks for confirmation of the size chosen. See question 8.

5. This seems to be a 5.25" disk.
Is this right? [Yes]

This question confirms that a 5.25-inch disk is being used. The next question is 6.

6. Enter name of disk, <HELP> for help, [UNKNOWN].

For the 5.25-inch disk, you must supply a recognized disk name or disk parameters. If you supply the name of a disk, the next question is 10. If you specify Unknown (the default), the next question is 7. To obtain a listing of known names, press the HELP key.

The disk.params file contains the name and parameters of various disks. If you wish, you may edit the file to add other disks (enter the information in exactly the same format as the existing entries). Thereafter when you run Partition, you can give the name in response to this question.

7. Would you like to enter the parameters yourself? [YES]

If you specify YES, questions 7a through 7e are presented. A NO response returns you to Question 6.

7a. No. of heads:

There is no default response to this question.

7b. No. of cylinders:

There is no default response to this question.

7c. Write precompensation cylinder:

The inner cylinders are more densely packed in terms of bits per linear inch, and whenever two bits are written in close proximity to each other, bit shift may occur. It may also occur, but to a lesser degree, because of phenomena such as random noise, speed variations, etc. The technique called write precompensation reduces bit shift by detecting which bits will occur early and which will occur late and writing these bits in the opposite direction of the expected shift. Supply a cylinder number.

7d. Boot size:

The size of the boot area is typically 32.

7e. Sectors per Track:

PERQ Systems typically supplies disks with 16 sectors per track, but the number may be different on the disk you are using.

The next question is 10.

8. Is this a nn-MByte disk? [Yes]

This question confirms the size of the Shugart disk. The default is the size of disk. In the actual question, the size (12 or 24) replaces nn above.

The next question is number 10.

9. Is this a Single (S) or Double (D) sided floppy?

If you specified the floppy in question 2, you must specify whether the floppy is single- or double-sided. You can partition a floppy that is formatted on both sides as either single or double sided. There is no default response to this question.

The next question is number 10.

10. Do you want to initialize the whole device? [No]

Respond Yes if you are starting from scratch (for example, on a newly formatted floppy or when installing the file system on a new machine). If you want to modify a device with existing file system structures, respond No.

If you respond Yes to this question, Partition asks for information about each partition in turn. See questions 21 through 27.

If you respond No to this question, Partition reads the Device Information Block to find the existing partitions and displays them in the order that the partitions appear on the disk. The program then asks for the modifications you wish to make. See questions 11 through 20.

The default response to this question is No (the program modifies, rather than initializes, the device).

11. Do you want to rename the device? [No]

This question appears if you responded No to question 10 and begins the device modification (rather than initialization) sequence. The first question in this sequence allows you to change the name of the device.

Before renaming a device, note that the linker incorporates the name of the device where a program was linked into the Runfile. Therefore, if you rename the device, you cannot run existing programs on that device. This includes the Shell, Login, and the Partition program itself.

If you respond Yes, Partition requests confirmation and then prompts for the new device name. See question 12.

If you respond No, the program asks which partition you want to modify. See question 13.

The default response to this question is No (to modify a partition, rather than rename the device).

12. New device name [current name]

    If you responded Yes to question 11 and to the subsequent request for confirmation, this question allows you to enter a new device name. Enter a device name from 1 to 8 characters. The Partition program writes the name you enter in the Device Information Block (DIB words 114 through 117).

    Note that the default response to this question is the current name of the device.

    After you enter the device name, the program asks whether or not you want the device remounted. See question 27.

13. Which partition do you want to modify?

    Enter the name of the partition you wish to modify. The program displays a summary of that partition's Partition Information Block and then prompts for the modification. See questions 14 through 16.

    There is no default response to this question.

14. Do you want to split this partition? [No]

    Since files cannot cross partition boundaries, it is never a good idea to split a partition that contains files. The system destroys any file that has blocks in both partitions. However, it is safe to split an empty partition or to split a partition that you plan to erase.

    If you respond Yes to this question, Partition requests confirmation and then prompts for the number of pages in each partition and the name of the new partition. See questions 15 and 16.

    If you respond No to this question, Partition continues with the modification queries. See question 17.

15. How many pages would you like in the first half?

    The response to this question specifies how much of the partition to leave with the old partition (the partition being split). The remainder of its blocks are put in the new partition. The Partition program then asks if you want to initialize the partition pages and thus create a new free list. See questions 24 through 26. After initializing the pages (or immediately, if you respond no to initializing pages), the program requests a name for the new partition. See question 16.

16. Name of new partition? Partition name (up to 8 chars):

    Enter the name of the new partition. The Partition program then asks if you want to initialize the pages for the new partition. See question 24.

17. Do you want to merge this partition with the next? [No]

    If you responded No to question 14, this question allows you to combine two partitions. It is safer to combine two partitions together than to split one apart.

    If you respond Yes to this question, the program joins the partition you specified in question 13 with the partition which is next on the disk. This is the only time the order of the partitions on the disk matters. Partition then asks if you want to initialize the new partition. See question 24. If you want to erase the new, bigger partition, respond Yes to question 24 to initialize the pages. To save all the files from both partitions, respond no to question 24. Then, after Partition exits, run the Scavenger program (described in Section 5.2) on the new partition. When running the Scavenger in this case, be sure to tell it to rebuild the directories so that the directories of the two partitions can be joined together.

    If you respond No to this question, Partition continues with the modification queries. See question 18.

18. Do you want to initialize this partition? [No]

    If you responded No to question 17, this question allows you to initialize the partition specified in question 13.

    A Yes response is a fast way to delete all the files in the partition.

    If you respond Yes to this question, Partition asks if you want to initialize the partition pages. See question 24. There are few reasons to initialize the partition without initializing the pages. However, if you respond No to question 24, you must use the Scavenger program to recreate the directory immediately after running Partition.

    If you respond No to this question, Partition continues with the modification queries. See question 19.

19. Do you want to change the partition name? [No]

This question allows you to change the name of the partition.

Before renaming a partition, note that the linker incorporates the name of the device and the partition where a program was linked into the Runfile. Therefore, if you rename the partition, no program in that partition can be run. Do not change the name of the partition that is used in the current path since this invalidates the default path name. Also, all entries in the search list that refer to the renamed partition will no longer work. After a rename, the system may not be able to find the Shell or any other programs.

If you respond Yes, Partition requests confirmation and then prompts for the new partition name. See question 20.

If you respond No, the program asks if you want the device remounted. See question 27.

The default response to this question is No (to exit the program, rather than rename the partition).

20. New partition name [current name]

If you responded Yes to question 19 and to the subsequent request for confirmation, this question allows you to enter a new partition name. Enter a partition name from 1 to 8 characters. The Partition program writes the name you enter in the Partition Information Block (PIB words 114 through 117).

Note that the default response to this question is the current name of the partition.

After you enter the partition name, the program asks whether or not you want the device remounted. See question 27.

21. Name for root partition (up to 8 chars):

This question begins the device initialization sequence if you responded Yes to question 10 (to initialize the entire device). The device must first be given a name (from one to eight characters).

The Partition program writes the name you specify in the Device Information Block (DIB words 114 through 117).

There is no default response to this question; you must enter a device name.

Partition then displays the number of logical blocks (pages)

available for the first partition (partition 0) and asks for the number of blocks to include in the partition. See question 22.

22. How many pages would you like in it? (0 => all)

The file system restricts device partitions to fewer than 32768 logical blocks (pages).

To permit the Scavenger (see Section 5.2) to handle a partition in one pass, each partition should have 10080 or fewer blocks.

The number of blocks per cylinder governs the size of a partition on a device. All partitions must be multiples of the number of blocks per cylinder for a given device. The minimum size for partitions is the number of blocks per cylinder, as follows: 12-megabyte Shugart and 35-megabyte Micropolis disk, 120; 24-megabyte Shugart disk, 240; and a floppy disk, 6. On 5.25" disks the number of blocks per cylinder will vary.

After you enter the number of blocks for a partition, the program requests the partition name. See question 23.

23. Partition name (up to 8 chars):

You must name each of the partitions (from one to eight characters). Examples of partition names used for the hard disk include "boot", "user", and "exp". A partition can have the same name as a device, but all the partitions must have unique partition names.

After you enter the name, Partition displays a summary of the PIB and asks if you want to initialize the partition. See question 24.

24. Do you want to initialize partition pages? [Yes]

This question begins the partition initialization sequence. Your response specifies whether or not the program initializes the partition's logical blocks (pages). If you are initializing the entire device (you responded Yes to question 10), Partition repeats this question for each partition on the device. If you are modifying a partition (you responded No to question 10) and you want to initialize it (you responded Yes to question 18), Partition asks this question once.

A Yes response to this question places all the logical blocks in the partition on the free list. If you are initializing the entire device (you responded Yes to question 10), you should respond Yes to this question. If you are modifying a

partition and initializing it (you responded YES to questions 11 and 18), it is recommended tha you respond yes to this question also. There are few reasons to initialize the partition itself without initializing the pages.

If you respond Yes to this question, Partition asks whether or not to test after initializing the pages. See question 25.

If you respond No to this question, you must use the Scavenger program (refer to Section 5.2) to recreate the directory immediately after running Partition.

The default response is Yes (to initialize the partition pages).

25. Do you want to test after initializing? [Yes]

Although testing after initializing slows the process somewhat, it is good practice to do this testing. If any logical blocks are found to be bad during testing, they are removed from the free list and thus never accessed again.

If you respond Yes, Partition asks if you want to write every page twice. See question 26.

If you respond No, Partition simply initializes the logical blocks and asks if you want the device remounted. See question 27.

The default response is Yes (to test after initializing).

26. Do you want to write each block twice? [Yes]

Writing every logical block twice provides some additional protection from bad pages; random data is written into the header and body of each block. In practice, some bad blocks on the disk pass the first test and fail this one.

27. Do you want the device remounted? [Yes]

After all the blocks on the device have been included in a partition or after the partition has been modified, the program displays some data about the device or partition and asks whether or not to remount the device. Only mounted devices can be accessed, so if you plan to use the device respond Yes.

The Partition program accepts a switch, /BUILD, on the command line, which partitions the entire device. Note that the arguments must be specified on the command line. This is a dangerous thing to do and it is only recommended for command files which bring up an entire disk. The format for this switch is

Partition/<DiskType>/Build <Dev> <DeviceName> <PartNameList>

where <dev> is either "H" for the hard disk or "F" for the floppy. Specify the device name next, followed by enough partition names for the entire device. The partition names are used in the order specified so the first name will be used for the first partition on the disk. All partitions except the last will be the standard size (10080 blocks). If extra partition names are specified, they are ignored.

If the device is already formatted, the program requires confirmation before erasing the device. If you respond No to the confirmation request Partition runs, asking all the questions.

5.2 THE SCAVENGER PROGRAM

The Scavenger program fixes a partition on a device that contains useful files. Run the Scavenger whenever an inconsistency is found in the file system or when some program aborts and instructs you to run it. The Scavenger checks all files for consistency, rebuilds the free list, and creates a new directory structure for the partition. The Scavenger also removes bad boots.

The Scavenger should be run only on devices initialized by the Partition program.

Do not type CTRL/C after the Scavenger begins writing on the device. (You can type CTRL/C during the read pass.) If you type CTRL/C after the Scavenger begins rebuilding the directory, you may not be able to access anything in the directory. If this happens, rerun Scavenger from another partition.

The Scavenger cannot recreate the directory if there are no free blocks in the partition. Therefore, if your partition is full, you must delete some files before running the Scavenger. If you cannot delete any files due to a bad directory and there are no free blocks, then you cannot rebuild the directory. In this case, you must initialize the partition, thus losing all files there. Fortunately, this is a rare occurrence.

The Scavenger program fixes one partition at a time. You can scavenge the current partition or scavenge partitions other than the one you are currently running in.

The Scavenger program has three separate phases. In phase one, it checks and updates some of the system information and deletes bad boots. If you define a boot with MakeBoot (see Section 5.3), but either the microcode or system code files have been deleted, the boot is known bad and the Scavenger deletes it. However, if a boot file is deleted and another created in the same place before the Scavenger is run, the boot appears valid but will not work.

In the second phase, the Scavenger checks the partition you specify for consistency. The program reads all blocks and generates a new free list in ascending disk order (the Scavenger discards the old free list). Also during this phase, the Scavenger marks as "bad" those blocks that are not readable. If the blocks marked as bad cannot be rewritten, they are marked as "incorrigible" and removed from the file system. All blocks that were allocated to files containing bad or incorrigible blocks are put in a bad file. Additionally, malformed chains are added to the bad file. You specify a name for this bad file in phase three.

During phase three, the Scavenger rebuilds the directories for the partition. You can direct the Scavenger to delete the old

directories, if desired. Otherwise, the old directories are marked as such and the program adds a "$" to the end of the directory name. The Scavenger aborts if there is not enough room in the partition to create copies of all the directories, leaving the directories only partially rebuilt. In this case, delete some of the files in the directory and then rerun the Scavenger.

Prior to entering a name in a directory or creating a new directory, the Scavenger validates the filename. If a bad name is found (the name includes a control character, "<", "/", ":", or is null) or two files have the same name, Scavenger requests a new filename. In addition, the name cannot end with a ">" or contain ">..>" or ">.>". After Scavenger is finished, you can examine the files with bad names to see whether they contain useful information. If so, rename or edit the files to recover the data. Otherwise, simply delete the files.

Also during the third phase the Scavenger checks the length of all files and allows you to specify a new length. Note that the length refers to the stored length (in the FIB) rather than the number of blocks in a file. Certain files, like directories and swap files, do not set the length field. File lengths usually become wrong when the file is opened and written, but not properly closed (for example, when a transfer is aborted).

To run the Scavenger program, simply type

Scavenger [partition]

in response to the default PERQ prompt.

If you omit the partition name, the Scavenger asks a number of questions before it begins processing the partition. Note that some of the questions have a default response.

If you include a partition name, the Scavenger uses all defaults and runs until completion unless there are any serious errors. If errors are discovered, the Scavenger requests confirmation before exiting. Note that the partition you specify must be on the hard disk.

Following is a list of the questions Scavenger will ask, with an explanation of each.

1. Which device to scavenge? (F = Floppy, H = Harddisk): [H]

   Your response to this question specifies the device that contains the partition to modify. The choices are the Hard disk or the Floppy disk.

   If you respond H (to indicate the hard disk), Scavenger checks to see what type of disk drive you are using and asks whether

your disk is a MICROPOLIS 8-inch drive, a Shugart 14-inch drive, or a 5.25-inch drive. See question 2 if you are running on a PERQ2, question 3 if running on a PERQ, and question 4 if using a 5.25-inch drive.

If you respond F (to indicate the floppy disk), Scavenger asks whether the floppy is single or double sided. See question 8.

2. This seems to be a MICROPOLIS 8-inch disk.
Is this right? [Yes]

This question confirms the disk choice.

The next question is number 9.

3. This seems to be a SHUGART 14-inch disk.
Is this right? [Yes]

This question confirms the disk choice. Scavenger then checks to see whether the disk is a 12- or 24-megabyte disk and asks for confirmation of the size chosen. See question 7.

4. This seems to be a 5.25" disk.
Is this right? [Yes]

This question confirms that a 5.25-inch disk is being used. The next question is 5.

5. Enter name of disk, <HELP> for help, [UNKNOWN].

For the 5.25-inch disk, you must supply a recognized disk name or disk parameters. If you supply the name of a disk, the next question is 9. If you specify Unknown (the default), the next question is 6. To obtain a listing of known names, press the HELP key.

The disk.params file contains the name and parameters of various disks. If you wish, you may edit the file to add other disks (enter the information in exactly the same format as the existing entries). Thereafter when you run Scavenger, you can give the name in response to this question.

6. Would you like to enter the parameters yourself? [YES]

If you specify YES, questions 6a through 6e are presented. A NO response returns you to Question 5.

6a. No. of heads:

There is no default response to this question.

6b. No. of cylinders:

There is no default response to this question.

6c. Write precompensation cylinder:

The inner cylinders are more densely packed in terms of bits per linear inch, and whenever two bits are written in close proximity to each other, bit shift may occur. It may also occur, but to a lesser degree, because of phenomena such as random noise, speed variations, etc. The technique called write precompensation reduces bit shift by detecting which bits will occur early and which will occur late and writing these bits in the opposite direction of the expected shift. Supply a cylinder number.

6d. Boot size:

The size of the boot area is typically 32.

6e. Sectors per Track:

PERQ Systems typically supplies disks with 16 sectors per track, but the number may be different on the disk you are using.

The next question is 9.

7. Is this a nn-MByte disk? [Yes]

This question confirms the size of the Shugart disk. The default is the size of disk. In the actual question, the size (12 or 24) replaces nn above.

The next question is number 9.

8. Is this a Single (S) or Double (D) sided floppy?

If you specified the floppy in question 1, you must specify whether the floppy is single- or double-sided. There is no default response to this question.

The next question is number 9.

9. Can I make changes to your disk [Yes]?

The response to this question specifies whether or not the Scavenger program can make changes to the device in the first two phases of the program (the Scavenger fixes directories later). This is similar to the debug option in the Partition program (refer to question 1 in Section 5.1).

If you respond No, the Scavenger checks the partition for errors and reports them, but does not fix anything. If you are running the Scavenger only to fix the directory, it is about twice as fast to respond No to this question; otherwise, Yes is a good idea.

The default is Yes.

The logical header of every block contains information about the state of that block. The information includes the count of the block in the file (the file relative LBN) and a two-word identifier for the file the block belongs to (the serial number). The next two questions allow you to check the correctness of these values.

10. Do you want logical block consistency checking [Yes]?

This question asks whether or not the Scavenger should check the Logical Block Number field in the block's logical header. Remember that this field specifies the file relative logical block number (the first, second, third, and so on block of the file).

If you respond Yes, the Scavenger compares the LBN field in the logical header with the random index for each file.

If you respond No, the Scavenger omits this test.

The default is Yes (to check the file relative logical block number).

11. Do you want serial number consistency checking [Yes]?

This question asks whether or not the Scavenger should check the Serial Number field in the block's logical header. The serial number of all blocks in a given file is the Logical Disk Address of the file's File Information Block.

If you respond Yes, the Scavenger compares the Serial Number field in the logical header with the LDA of the FIB.

If you respond No, the Scavenger omits this test.

The default is Yes on machines with more than 256k bytes of main memory and No on 256k byte machines.

If the Scavenger continually aborts due to FullMemory, respond No to either question 10 or question 11. The default response of No for serial number checking on machines with 256k bytes of main memory avoids the FullMemory condition; simply press return to get the default answer.

12. Is there enough room to do it in one pass [Yes]

    This question asks if there is enough memory to do the scavenge in one pass. If your partition has 10080 or fewer blocks in it and if the screen has been shrunk (the Shell shrinks the screen when it knows you are running the Scavenger), then respond Yes. If your partition is larger than 10080 blocks, respond No.

    If you respond No, the Scavenger uses three passes, which correspondingly slows the program.

13. How many tries for a suspect read? [1]

    This question asks for the number of retries for a suspect read. The default is 1.

If you responded Yes to question 9 (whether or not the Scavenger could change your disk), the Scavenger asks questions 14 through 18.

If you responded No to question 9, the Scavenger asks question 19.

14. Do you want temporary segments deleted [Yes]?

    This question asks whether or not the Scavenger should delete temporary files. Temporary files exist for swapping; all user files are permanent.

    The recommended response is Yes, which is the default.

15. Do you want old bad segments deleted [Yes]?

    This question asks whether or not the Scavenger should delete old bad segments. Bad segments contain files with incorrigible blocks from a previous Scavenger run.

    If you respond Yes to this question, then Scavenger adds the bad file created by the previous Scavenge of this partition to the free list.

    If you respond No to this question, Scavenger retains the old bad segment.

16. Can I rewrite bad blocks [No]?

    This question asks whether or not Scavenger can rewrite bad blocks. If a block cannot be successfully read in the specified number of retries (see question 13), writing new data onto the block could fix the problem. However, because writing new data has only a small likelihood of fixing the problem, the default response is No.

If you respond Yes, the Scavenger writes new data in the bad blocks. If the write or a subsequent read fails, then the block is "incorrigible," otherwise it is "bad."

If you respond No, the block is marked "incorrigible" as soon as the read fails. If there are only transient read errors, the block is left alone.

17. Type pairs to ignore (cyl head cyl head ...): []

This question permits you to enter cylinder/head pairs that you assume are bad.

If you suspect a bad cylinder or track, enter the cylinder number followed by a space followed by the head number. Continue this sequence for the suspect cylinder/head pairs. Note that a carriage return signifies end of input. Therefore, if the bad pairs require more than a single line for input, simply wrap the pairs to the next line; do not press carriage return until you enter all the bad pairs.

The default ignores no cylinder/head pairs.

18. Type other blocks to ignore: []

This question permits you to enter block numbers that you assume are bad.

If you suspect a bad block, enter the Logical Block Number of the bad block. You can enter up to 15 bad blocks, separated by spaces. Again, carriage return signifies end of input. If you suspect that more than half the logical blocks are bad on a track, writeoff the entire track. Disks have the following number of blocks per track: Shugart (either a 12M or 24M), 30; Micropolis, 24; 5.25" disk, 16; and floppy, 6.

The default ignores no logical blocks.

Scavenger then asks whether or not you want complete error listing; see question 19.

19. Do you want complete error listing [Yes]?

This question permits you to specify whether or not you want notification of individual errors.

Respond Yes for complete error listing or No for a summary of errors. Yes is the default.

After you answer the above questions, the Scavenger begins fixing the partition.

The Scavenger updates the title line of the window to show what it is working on and uses various cursors to show progress of the different passes.

First, the Scavenger fixes discrepancies if you allowed changes (responded Yes to question 9). If the Scavenger finds a problem with the partition or device information blocks it cannot fix, it asks for help. The problem could be that you specified the wrong device type in question 1 or that the device is not a file system device (for example, an RT-11 format floppy or a hard disk that has not been initialized by the Partition program). These are easily remedied. However, the problem could be that the device discrepancies are beyond repair. Unfortunately, the only fix in this case is to re-partition the device from scratch.

After the device and partition information checks out, Scavenger displays a summary of the Device Information Block (including partition names) and asks which partition it should work on.

20. Which partition do you want to scavenge?

    Enter the name of the partition to scavenge. There is no default response to this question.

When you enter the partition name, the Scavenger displays a summary of the specified partition's Partition Information Block. Note that the values are those stored in the PIB prior to the scavenge.

The Scavenger now makes a read pass through the partition, building tables of each block's next and previous link (this is the data usually visible in the lower portion of the screen). The pass is done in eight parts for efficiency; the cursor goes down the screen eight times before the next step.

The Scavenger then checks the tables built during the read pass for consistency. The cursor changes to show that consistency checking is in progress. If any loops are found, the Scavenger breaks the loops and blinks the screen to show that a loop has been fixed.

Following the consistency check, the Scavenger rebuilds the free list. Note that the Scavenger rebuilds the free list only if you responded Yes to 9 (you permitted changes to the disk). Rebuilding the free list requires a write pass for all blocks on the free list. The Scavenger displays a write cursor. If any bad blocks were found, some more reads and writes are necessary to make the bad blocks into a well-formed chain.

At this point, the Scavenger writes the new Partition Information Block and then begins the directory building pass. The Scavenger asks whether it should rebuild the directory.

21. Do you want to rebuild the directories?

This question permits you to rebuild the partition's directory structure.

Sometimes the Scavenger recommends that you do this, in which case the default is Yes. Otherwise there is no default.

If the Scavenger recommends that you rebuild the directories or if you suspect there is a problem with the directories and there are enough free blocks, respond Yes.

If you respond Yes, the Scavenger asks if you want to delete the old directories (see question 22).

22. Delete old directories [Yes]?

If you responded Yes to question 21, this question permits you to delete the old directories.

If you respond Yes, the Scavenger deletes the old directories.

If you respond No, the Scavenger saves the old directories for later inspection; the Scavenger appends a dollar sign ($) to the end of the old directory names and changes their file type to ExDirFile (directories all have the type DirFile). The old directories are just files that you can delete after the scavenge.

The Scavenger then creates new directories as needed. Note that empty directories are not recreated.

Since the directory reappears after a scavenge, the directory rebuilding process makes it easy to recover from overwriting or deleting a directory.

The Scavenger checks and allows fixing file lengths if desired. For each file, it checks the stored length with the actual number of blocks in the file. If they do not match, it allows you to specify a new stored length. This can be any value, but making it bigger than the number of blocks in the file is not recommended. The default for the stored length is the number of blocks in the file. The Scavenger does not check the lengths for directory files or files with their type field set to "SWAPFILE."

Each file's random index allows the file system to find a random logical block without searching down the chain from the file start. The Scavenger, as part of the directory building phase, can rebuild the random indices for all files. There is a separate question for this with a default answer of No. There is usually no reason to rebuild the indices unless Scavenger asks you to. Building the random indices for large files takes a long time.

If a bad file was created, the Scavenger asks for a name for the file at the end of the directory building phase. If you allow Scavenger to enter and fix the indices for the bad file, you can then type and edit it as a normal file. In this way, you can reclaim some useful information.

5.3 MAKEBOOT

The MakeBoot program creates new systems. Any program can be made to work "stand-alone" (so it can be booted) by initializing various modules as the System program does. It is generally not necessary or desirable to have programs other than the System be stand-alone.

If you follow the instructions of this section and barring errors and bugs in your modifications, you should be successful in making a new system.

To help you avoid errors that will make your system non-bootable, observe the following precautions:

Back up important files on floppy disks before you begin changing the PERQ Operating System.

Maintain at least one partition on the disk that runs the old system. This enables you to boot the PERQ in the event that your new system contains bugs. When you receive your PERQ, the hard disk is divided into several partitions. At least one partition contains a pair of boot files: System.<n>.<x>.Boot and System.<n>.<x>.MBoot. The <n> in the file name is the current system version number, and the <x> in the file name represents the character that you hold down to use the corresponding boot files and partitions. The .Boot file contains the Operating System, and the .MBoot file contains the QCode interpreter microcode.

Do not change the default boot files System.<n>.a.Boot and System.<n>.a.MBoot until your new system is completely debugged.

Create a new directory or select an unused partition for your new experimental files when you begin making your changes to the system. Copy the sources of the files you want to change into this area before you begin editing. Compile all new .Seg files into this area. Do not use the root directory in the default partition--the one that is entered by the default boot letter ("a" is default--the same as not holding down a key). The default boot files are System.<n>.a.Boot and System.<n>.a.MBoot.

If you do not change the default boot files or the files in the root directory of the default partition, you will still have source and binary files that you can fall back on.

Creating a new system usually consists of the following steps:

1. Evaluate the change you intend to make;
2. Create a directory to work in;
3. Edit and compile system modules;
4. Edit and compile system programs;
5. Link the system and system programs;
6. Prepare the system configuration file;
7. Write a boot file;
8. Test the new system;
9. Iterate at step 3.

Before you begin, determine how extensive the changes are. If you are changing the definition of data structures known by the microcode (for example, memory manager tables) or data structures that live across boots (for example, structures on disk), changing the existing exports of modules that the compiler knows about (Code, Dynamic, and Stream), or changing the format of .Seg files, you need to do a complicated bootstrapping operation which is beyond the scope of this manual. The following criteria tell you how much you need to change.

> Are you changing the existing exports of any system modules? If not, you need only re-compile those modules that you change. If so, you may need to re-compile those modules and programs that import the ones you are changing.
>
> Are you adding exports but not changing any that already exist? If you don't change existing exports, you need re-compile only those modules that you change. However, you must add your new exports at the end of the export list. By adding at the end of the export list, you do not change the storage allocation of existing variables or the routine numbers of existing procedures and functions. If you change either of these, you must re-compile all modules and programs that import the ones you are changing.

Use the MakeDir utility to create a new directory for your experimental files. This new directory should be in the partition which contains the old system (usually, the Boot partition). Copy sources of the system modules and programs into this directory. You may choose instead to work in a partition which, up until now, has not been used. Using a new partition is somewhat safer than merely creating a new directory in some old partition.

Edit the modules and programs that you need to change. Re-compile those modules and programs that you have changed and any others indicated by your evaluation of the changes.

Once all necessary changes and compilations have been done, link the new system and system programs. Choose a new system version number. PERQ Systems Corporation intends to use the version numbers between 1 and 99 for releases of the official PERQ Operating System. Avoid these numbers to prevent conflicts with future PERQ System's releases. For example, choose version number 100 for your new system.

The new run files for System, Login, Shell, and Link should be in the root directory of the partition which contains your new system. You should use the following link commands to link your new system (assuming that the path is set to the partition that contains your files):

```
Link System~System.100/System
Link Login~Login.100
Link Shell~Shell.100
Link Link~Link.100
```

After creating the new system, MakeBoot checks the partition's root directory for Login.Run, Shell.Run, and Link.Run and prints a warning message if one or more of these files is not found.

Before you write the boot file, create a system configuration file which describes the swappability of modules in the system. You can probably copy the configuration file for the current version of the operating system. The default configuration file is named System.<n>.Config. If you must change the swappability of modules in the system, you can copy the old file and edit it.

You are now ready to write a boot file. Before you run MakeBoot, choose a boot-letter that is not already in use for this new system. You can use Details/Boots to find out which letters are in use. After choosing a boot letter, run MakeBoot. The MakeBoot program asks a series of questions. Following are the questions MakeBoot will ask, with an explanation for each.

Root file name:

> The run file name given to MakeBoot determines on which device and partition the boot will be. MakeBoot takes the directory part of the file name and uses that to determine on which device and partition to put the boot. Therefore, to make a boot somewhere, first copy the run file to that partition. After specifying the run file, Makeboot asks for the configuration file.

Configuration file name [System.<n>.Config]:

> The default configuration file is named System.<n>.Config.

The configuration file tells MakeBoot which System modules are swappable. Each line in the file describes the swappability of a single module in the form:

<module name> <swappability>

The <swappability> is chosen from the following:

SW - module is swappable
US - module is not swappable, but may be moved in memory
UM - segment is neither swappable nor movable

The default for code segments (modules) is US. Thus, you only need list the modules in your system that you want swappable or unmovable.

Names with asterisks are recognized as special module names. They are chosen from the following list:

SAT - Segment address table (default UM)
SIT - Segment information table (default US)
Cursor - Display cursor (default UM)
Screen - Display screen (default UM)
Font - Character set (default US)
Stack - Run-time stack (default US)
Names - System segment names (default SW)
IO - Input/output tables (default UM)

Do not change the swappability of the special segments and, unless you are sure you know what you are doing, do not change the swappability of existing system modules.

System data segments that the hardware or microcode uses cannot be moved, most data used by the operating system (*SIT* and *FONT*) cannot be swapped, and the code that makes up the swapping system itself cannot be swapped. Since the default for code segments is US, you should add entries to the configuration file if you add modules to the system.

The next question asks for the boot character.

Which character to boot from?

MakeBoot creates a boot file by associating a stand-alone run file (such as a system) with a letter. The lower case letters are assigned to the hard disk and the upper case letters are assigned to the floppy disk. The default when no keys are held down is lower case "a". You can free boot letters of the associated boot by deleting the system and/or interpreter boot files.

The next question asks whether or not to write the boot area.

Do you want to write the boot area [No]:

The boot area of the disk contains a microprogram which runs diagnostics and reads the .Boot and .MBoot files. Regardless of how many boot letters are defined for a device, there is only one set of boot microcode (Vfy.Micro and SysB.Micro). Therefore, unless you are putting the first boot on the device or you are modifying Vfy.Micro or SysB.Micro, you need not write the boot area.

There are two files associated with each boot letter. The system boot file is Pascal and the interpreter boot file is microcode. The next questions ask if you want to write a system boot file and an interpreter boot file.

MakeBoot puts the output boot files wherever you specify, but it is important that the interpreter and system boot files be in the same partition. The device and partition in which the boot file is created will be the default path after the boot. This means that there must be at least a "Login.nn.run" and a "Shell.nn.run" (where "nn" is the version number of the system run file), in the root directory of the partition. It doesn't matter if the boot files are in a subdirectory in the partition; the run files mentioned above must be in the Root directory.

Write a system boot file [Yes]:

MakeBoot creates the system boot file by reading the supplied run file.

If you respond Yes, MakeBoot asks for the name of the system boot file.

If you respond No, MakeBoot asks for the boot file to copy.

The system code includes the default character set font, Fix13.Kst. MakeBoot looks for the default font in all the search paths. If not found, MakeBoot asks for the name of the character set.

Enter name of character set [Fix13.Kst]:

You can specify a character set which is different than the standard (Fix13.Kst). If you use a non-standard character set, some programs (like the Editor) may not work properly. For the Editor and certain other programs to work, the default font must be fixed width and thirteen bits high and nine bits wide.

Make the screen be (screentype) [Yes]:

The landscape screen uses a larger screen segment than the portrait screen. This question specifies the type of screen to be used with the new system and thus the size of the new system's screen segment. In the actual question, Portrait or Landscape replaces (screentype), depending on the screen of the machine.

The default is Yes (to make the screen segment the size required for the machine you are running on).

Write an interpreter boot file [Yes]:

Usually, all boot files use the standard microcode. MakeBoot uses the standard microcode to create the interpreter boot file if you so specify.

If you respond Yes, MakeBoot writes the boot file containing the microcode which is the Q-machine interpreter. Unless you are changing the interpreter microcode, you need only write this part once for a given boot letter. Note that you may add other microcode files to the boot file (as long as they do not overlap the standard microcode).

If you have already created a boot for the current letter and you have not changed microcode, it is not necessary to make a new interpreter boot file, but it never hurts to do so. If you want to load the standard microcode and it is found by MakeBoot, type CR when it asks for an interpreter microcode file.

Write a Z80 load boot file [Yes/No]:

On a PERQ2 machine, a special file (.ZBoot) provides GPIB, pointing device, RS232, and clock support. This question asks whether or not you wish to write this file.

The default is Yes if MakeBoot fails to find an existing .ZBoot file and No if MakeBoot finds an existing .ZBoot file.

If you respond Yes, MakeBoot asks for the name of the new Z80 boot file and the file name from which the .ZBoot file is to be created.

Note that MakeBoot never writes a .ZBoot file on a floppy.

You are now ready to boot the new system and test it. Hold down the boot key you selected and press the Boot button. If all goes well, your new system will announce itself.

Note that when you try to run most programs, the loader informs you that they were linked under the old system. This means you must re-link them for the new system. It is a good idea to create another new directory to hold these run files. This avoids conflicts when you run the old system. You can make a Login profile to add this directory to your search list when you log in under the new system.

Once the new system is running, link the system utility programs. Set your path to the new directory that you created to contain the run files. Push the directory containing the old .Seg files onto your search list, and then push the directory containing the new .Seg files. Now, type:

```
Link ProgramName
```

for each utility program you wish to link.

Re-compile any programs that import modules whose exports have changed.

If the system doesn't come up, you can look at the diagnostic display to determine where in system initialization the new system hangs.

Once the new system is debugged and working, you can use MakeBoot to rewrite the boot files associated with other boot letters. Before you rewrite the old boot files, be sure that some partition contains all files that make up the new system. This includes files that you have not changed. If you fail to make a partition containing all source, binary, and run files, you run the risk of deleting portions of your new system when you delete the old system.

The MakeBoot program accepts a switch, /BUILD, on the command line. If it is invoked with the /BUILD switch, you must specify all arguments on the command line. MakeBoot does not ask questions when you specify /BUILD. The format for this switch is:

```
MakeBoot [<dir>]System.<nn>/Build <bootKey>[/<disktype>]
```

where <dir> is an optional directory, <nn> is the system version number and <bootKey> is the character to boot from. The optional switch allows you to specify which SysB file to write: /CIO writes CIOSysB, /EIO writes EIOSysB, and /EIO5 writes EIOSysB for any kind of a 5.25" disk. The switch selects the correct IO microcode, allowing you to write the SysB file for a different type of machine. For example, if you are running on a PERQ but creating a system to run on a PERQ2, you would specify the /EIO switch. Makeboot then uses the default answers for all questions.

The following is an example that creates a new system. In the

example:

Underlined text is MakeBoot output
Commentary is given inside { }
The symbol <CR> indicates the RETURN key with no text
Assume you have chosen the boot letter "z"

>MakeBoot

Root file name: System.100

Configuration file name [System.100.Config]: <CR>

Which character to boot from? z

Do you want to write the boot area [No]: <CR>

{ The boot area of the disk contains a microprogram which runs diagnostics and reads the .Boot and .MBoot files. You need to rewrite this only if you are making modifications to Vfy.Micro or SysB.Micro. }

Write a system boot file [Yes]: <CR>
Enter name of new system boot file [System.100.z.Boot]: <CR>
Existing boot file to copy (return builds a new one): <CR>
Enter name of character set [Fix13.Kst]: <CR>
Make the screen be portrait [Yes]: <CR>

{ This writes the boot file containing the Pascal part of the system and special system segments such as the segment tables, the cursor, and the character set. Note that you may specify a character set which is different than the standard (Fix13.Kst). If you use a non-standard character set, some programs (like the Editor) may not work well. }

Write an interpreter boot file [Yes]: <CR>
Enter name of new micro boot file [System.100.z.MBoot]: <CR>
Existing boot file to copy (return builds a new one): <CR>
Use standard interpreter microcode files? [Yes]: <CR>
Interpreter microcode file: <CR>

{ This writes the boot file containing the microcode which is the Q-machine interpreter. Unless you are changing the interpreter microcode, you need only write this part once for a given boot letter. Note that you may add other microcode files to the boot file (as long as they do not overlap the standard microcode). }

Write a Z80 Load Boot file [Yes/No]: <CR>

{ On a PERQ2 machine, a special file (.ZBoot) provides GPIB,

pointing device, RS232, and clock support. This question asks whether or not you wish to write this file.

The default is Yes if MakeBoot fails to find an existing .ZBoot file and No if MakeBoot finds an existing .ZBoot file.

If you respond Yes, MakeBoot asks for the name of the new Z80 boot file and the filename from which the .ZBoot file is to be created.

Note that MakeBoot never writes a .ZBoot file on a floppy. }

Enter name of new Z80 boot file
[Sys:Server>system.22.x.ZBoot]: <CR>
Enter file from which ZBoot file is to be created
[System.ZBoot]: <CR>

5.4 FIXPART

FixPart is an experimental program for fixing the Device and Partition information blocks.

It is NOT recommended that you try to use FixPart without assistance.

Unlike the other file system utilities described in this chapter, FixPart is only partially automatic and can cause extensive damage to your system.

FixPart is currently the only way to fix bad partition and device information blocks. Fortunately, it is very rare that the device and partition information blocks require repairs that necessitate use of FixPart.

Always run the Scavenger (see Section 5.2) before you run FixPart; an apparent DIB or PIB problem may in fact be elsewhere and can be fixed by other means.

Once you are certain that you must run FixPart and you have qualified assistance, type

FixPart

The program first asks if you are sure you want to run FixPart. FixPart then asks for the device type. If you are using a 5.25-inch disk, the program will ask a series of questions about the disk. It then goes through and checks each partition for consistency with the other partitions and with the Device Information Block. If a name is dubious, FixPart asks if the name is valid. After all partitions are checked, FixPart displays a

summary of the errors.

If no errors were found, FixPart exits.

If errors were found, FixPart allows you to specify new start and end addresses for the partition and fix the names.

If the Device Information Block is not writeable, you must reformat the entire device and, unfortunately, lose all the data on the device.

If one of the Partition Information Blocks is not writeable, you may be able to save some information. First, run the Partition program and merge the partition with the bad information block to the partition before (see Section 5.1). You can then run the Scavenger to salvage the data (see Section 5.2). Note that if the first Partition Information Block is not writeable, this process does not work; you must reformat the device.